1981

Petri net models of program execution in data flow environments

Steven Fletcher Jennings
Iowa State University

JENNINGS, STEVEN FLETCHER

PETRI NET MODELS OF PROGRAM EXECUTION IN DATA FLOW ENVIRONMENTS

Iowa State University PH.D. 1981

PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark __✓__.

1. Glossy photographs or pages _____
2. Colored illustrations, paper or print _____
3. Photographs with dark background _____
4. Illustrations are poor copy _____
5. Pages with black marks, not original copy _____
6. Print shows through as there is text on both sides of page _____
7. Indistinct, broken or small print on several pages _____
8. Print exceeds margin requirements _____
9. Tightly bound copy with print lost in spine _____
10. Computer printout pages with indistinct print ✓
11. Page(s) _____ lacking when material received, and not available from school or author.
12. Page(s) _____ seem to be missing in numbering only as text follows.
13. Two pages numbered _____. Text follows.
14. Curling and wrinkled pages _____
15. Other _____

Petri net models of program execution in data flow environments

by

Steven Fletcher Jennings

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Major: Computer Science

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Iowa State University
Ames, Iowa

1981

TABLE OF CONTENTS

CHAPTER I. INTRODUCTION

The computer systems of the eighties are expected to be designed using powerful low-cost distributed parts to achieve increases in computing power and concurrency. Rapidly decreasing production costs of LSI and VLSI components facilitate this approach to the design process. The potential for high degrees of concurrency (and therefore reduced "real time" requirements for the execution of computer programs) have led many researchers to develop strategies for the efficient utilization of these distributed components. One such class of architectures, based on the concept of data flow, is designed to exploit the inherent parallelism within a program.

Conventional computer systems (those based on the von Neumann concept of computing) are characterized primarily by a centralized control and are driven by one or more instruction streams. Any parallel activity is explicitly denoted by the programmer. In contrast, data flow systems are designed to be driven by the availability of data. The traditional sequencing constraints are removed and an operation is enabled for execution as soon as its operands are available [Arvind and Gostelow 1977, Chamberlin 1971, Davis 1978, Dennis and Misunas 1975, Plas, et al. 1976]. There have been several architectures proposed for

data-driven machines and at least two prototypes under construction [Arvind and Gostelow 1977, Davis 1978, Dennis 1974, Dennis and Misunas 1975, Johnson, et al. 198Ø, Mago 198Ø, Plas, et al. 1976, Rumbaugh 1977, Watson and Gurd 1979]. These vary as to the level of intended exploitation of parallelism as well as the techniques to insure safe run-time environments for the transferral of data values.

The complexity resulting from the interaction and communication between numerous parts of distributed computer systems requires new methods for their analysis. Data flow systems containing many processing elements (functional units) fall into this category. The execution time behavior of programs run on these systems is especially relevant to the overall evaluation of proposed systems as well as determining the appropriateness of a given system for certain problem classes. Furthermore, the evaluation of an algorithm and its program representation is highly important in achieving high degrees of concurrency and resource utilization. Graph models of computations prove to be very amenable to the analysis of data flow programs, but either previous research has been limited to conventional high level constructs or proposed methodologies have proven to require excessive quantities of time to perform the analysis [Martin and Estrin 1967, Martin and Estrin 1969, Oldehoeft, et al. 1979, Ramchandani 1974, Retnadhas 1978, Retnadhas

1979]. Other researchers have used simulation for the performance evaluation of programs to be executed on parallel architectures [Gostelow and Thomas 1980, Lee 1980].

This research presents a method for approximating the time required to execute a data flow program (assuming adequate computing resources). This method is applied to the static program graph at compile time and yields a parameterized equation for the execution time performance of the program. Parameters include expected number of loop iterations, expected lengths of data streams, and expected probabilities of the outcome of conditional evaluations. One important aspect of this research is the reduction of nodes found in the program graph. Without these reductions, the time required to perform the analysis would be greatly increased. Reductions are made upon abstract operations found in the high level program. By making a few simplifying assumptions, the method is recursively applied to these abstract operations in the program graph using a top-down approach. Petri net representations of the abstract operations are analyzed to yield values that are combined with more traditional approaches in the reduction of the program graph. While this approach may introduce approximations at each stage, the major benefit is a significant reduction over other techniques in the time required for the analysis.

A high level language is introduced with many characteristics suitable for programs to be executed on data flow computers. Special emphasis is placed upon stream-oriented computations.

Of the many proposals for data flow architectures, two are treated in this research: the feedback model similar to the Dennis-Misunas architecture [Dennis and Misunas 1975] and the stream-oriented model similar to the Dennis-Weng architecture [Dennis and Weng 1979]. In the feedback model, an operation contains space for each operation. This operand space is reused should the operation be re-enabled. Therefore, to prevent overwrite of data values, feedback signals from successive operations are required for an operation to be enabled. In the stream-oriented model, language restrictions prevent the re-enabling of the same static copy of an operation and hence the analysis is greatly simplified. Iterative computations are performed recursively and data streams are implemented as linearized structures.

Two architectural models are used in this dissertation, but neither is promoted as an ideal architecture. The first is used since software simulation was available to validate the model [Oldehoeft, et al. 1978]. It has many features which would perhaps preclude it as a viable real machine architecture. The second, due to Dennis-Weng, is modeled

since it provides powerful stream facilities and has a reasonable chance for implementation as a real machine. Rather than promote architectures, the technique of analysis by use of Petri nets is promoted and is the subject of this dissertation.

Outline of Dissertation

Chapter II introduces a high level language for the presentation of this research. Chapter III describes Petri nets in general and a subclass, state machine decomposible Petri nets, in particular. The recursive reduction of the nodes found in a high level program is the subject of Chapter IV. Chapter V presents the analysis of five programs along with their simulated results. The extension to other architectures is discussed in Chapter VI which is followed by the concluding remarks of Chapter VII. Three appendices contain the templates for high level stream operations, the simulation programs discussed in Chapter V, and a description of the notation used in this work.

CHAPTER II. HIGH LEVEL LANGUAGE

This chapter describes the features of the high level programming language used in the presentation of this research. This language contains constructs considered representative of current languages designed for parallel machine environments and is devoid of several features incompatible with a functionally descriptive language; e.g., GOTO's, global references, and artificial sequencing of statements. Data flow systems are value-oriented and hence an appropriate language supports this orientation. The syntax of the language is loosely defined since it is less important in this work than the compatibility of the underlying semantics with the supporting architecture. The procedure orientation and some of the traditional constructs of the language to be described may be expressed syntactically in a different manner in future high level languages. The high level language described here is not considered to be a proposal for a new language but is only presented to facilitate this research.

The basic data types of integer, real, and boolean are supported along with a representative set of base level operations including arithmetic, logical, and relational operations. Furthermore, integer and real arrays are present with the operations "select" and "append." Any

identifiers declared of the above types are referred to as "scalars," including array identifiers, since their value (the array structure in the case of array identifiers) may be transmitted along a path in the program graph as a single token (or by a single reference). A sequence of scalars of a homogenous data type constitutes a stream data type. Identifiers may be declared as a specific stream type. The sequence of values forming a stream is terminated by a special final value, end-of-stream (eos), where an empty stream contains only the eos token. The full set of operations defined on scalars is extended to operate on elements of streams as appropriate. For example, the expression A + B, where A and B are of integer stream type, produces as a result an integer stream where the elements of A and B have been pairwise added. Expressions involving streams are valid only if the length of the streams involved in a given operation are equal. Any operation with eos tokens as operands produces an eos result.

All high level operations produce a multi-expression as a result where the arity of the expression is the same as the expected number of values used in the invoking statement [Ackerman and Dennis 1978]. This language supports the naming of expressions through the use of an assignment statement. Assignment is viewed as a definition, since in general the language is single assignment [Chamberlin 1971].

The language also includes blocks, conditionals, while-do constructs, procedure definitions and invocations, and streamed computations. Blocks are constructed with BEGIN-END pairs and may contain local declarations. Following the END statement, in parentheses, is the definition of the "out set" of the block: those values produced as the result of the evaluation of the block. For example, the block in Figure 2.1 produces a binary multi-expression.

```
BEGIN
REAL x,y,r,s;
x,y = a+b, a-b;
r,s = x*y, x/y;
END (r,s)
```

Figure 2.1. Sample block

The conditional takes the form:

```
IF <BOOLEAN EXPRESSION> THEN <BLOCK 1>
                        ELSE <BLOCK 2>
```

Depending upon the value of <BOOLEAN EXPRESSION>, either <BLOCK 1> or <BLOCK 2> is evaluated to produce the results of the IF statement. Needless to say, arity(<BLOCK 1>) = arity(<BLOCK 2>) = arity(IF). The <BOOLEAN EXPRESSION> produces a scalar result. The situation where the programmer would like to produce a boolean stream result is handled syntactically by the high level "SELECT" stream

operation (described later in this chapter).

Iteration is supported by the while-do construct which takes the form:

```
WHILE <BOOLEAN EXPRESSION>
   INITIAL <ASSIGNMENTS> DO
   <LOCAL DECLARATIONS>
   <BODY>
END (<OUTSET>)
```

The assignments in the initial clause specify initial values for the identifiers in the input set of the while-do. As an example, the while-do in Figure 2.2 computes $\sum_{i=1}^{10} i$.

```
WHILE  i <= 10
   INITIAL i = 1, n = Ø  DO
   INTEGER i,n;
   NEW  n = n + i;
   NEW  i = i + 1
   END (n)
```

Figure 2.2. Sample while-do

Procedure definitions take the form:

```
PROCEDURE <NAME> (<DECLARATION OF INPUT ARGUMENTS>)
   RETURNS (<DECLARATION OF OUTPUT PARAMETERS>)
   <LOCAL DECLARATIONS>
   <BODY>
   END
```

Procedures may be declared within any other procedure and may be recursively invoked. Only the normal entry and exit are allowed to a procedure. A procedure is invoked by supplying arguments of the procedure name and a procedure produces a multi-expression as a result. Side effects are

not possible. Only (and all of) the output parameters are associated with a unique value upon termination of a procedure. Procedures might just as well be called functions. Figure 2.3 contains an example of a procedure to calculate the factorial of its argument.

```
PROCEDURE fact (INTEGER n)
   RETURNS (INTEGER f)
   f = IF n > 1 THEN BEGIN
                        INTEGER x;
                        x = n * fact(n - 1)
                        END (x)
                ELSE BEGIN
                        INTEGER y;
                        y = 1
                        END (y)
   END
```

Figure 2.3. Sample procedure

A streamed computation is declared by the following syntax:

```
STREAMED
<LOCAL DECLARATIONS>
<BODY>
END (<OUTSET>)
```

Streamed identifiers may only be declared within streamed procedures. The declaration of streamed computations may prove to be syntactically unnecessary, but it appears in this work to facilitate the reduction of a streamed computation node. When it is desirable to perform a high level scalar operation (other than one of the base level

operations) upon each token of a stream, it is necessary to make the association between the appropriate scalars of the high level operation and the scalar tokens of the stream. For this reason, associations may be made within a streamed computation between "scalar" identifiers and stream identifiers. This association is made via an ASSOC entry in the local declarations of the streamed computations. After this association is declared, the interpretation is made that the scalar, when used in right hand context (referenced) in a construct, will take on successive values thereby reinvoking the construct once for each data token of the stream. Likewise, when the scalar is used in left hand context (assigned), the interpretation is made that the sequence of values assigned to the scalar constitutes the stream with which the association is made. The compiler must generate code for the appropriate handling of the eos tokens. For example, Figure 2.4 contains portions of a streamed computation in which the association between several scalars and several streams is made. Though it is possible for a procedure to have stream arguments, the procedure P in this example is defined to take a scalar as an input argument; hence, the procedure is reinvoked once for each token of the stream A.

Built-in routines are available for use by the programmer. Trigonometric functions, SQRT, and ABS are

```
STREAMED
   INTEGER a,b,c,d;
   INTEGER STREAM A,B,C,D;
   ASSOC a IN A;
   ASSOC b IN B, c IN C, d IN D;

   PROCEDURE P (INTEGER x) RETURNS (INTEGER y)

      ---

      END

   A = <stream creation>

   b = WHILE i < a
          INITIAL i = 1, n = Ø DO
          INTEGER i,n;
          n = n + i;
          i = i + 1;
          END (n);
   c = P(a);
   d = STREAMED
          (a used in right hand context)
       END

     ---

   END
```

Figure 2.4. Associations in a streamed computation

defined for integers, reals, integer streams, and real streams in addition to a wide range of high level stream operations. The stream operators will be described in more detail in a subsequent chapter but consist of the following:

a) FIRST (stream)

b) REST (stream)

c) CONS (scalar, stream)

d) EMPTY (stream)

e) SUM (stream)

f) PROD (stream)

g) MIN (stream)

h) MAX (stream)

i) SIZE (stream)

j) BUF (stream)

k) UNBUF (array)

l) REPL (scalar, stream)

m) SCREATE (scalar1, scalar2, scalar3)

n) SELECT (stream1, stream2, stream3)

o) SUBSTM (stream, scalar1, scalar2)

The FIRST operation yields the first token of its argument stream as its result. The REST operation yields its stream argument without its first token as its result and hence is undefined on the empty stream. The CONS operation places its scalar argument in front of its stream argument to yield its stream result. EMPTY produces a boolean result depending on whether its argument is the empty stream or not. SUM, PROD, MIN, MAX, and SIZE produce appropriate scalar results based on their argument streams. The BUF and UNBUF operations are used to buffer and unbuffer streams to and from, respectively, an array. The REPL operation produces a stream the same length as its stream argument consisting of the same scalar value for each data token. SCREATE produces an integer stream of values from

scalar1 through scalar2 by scalar3. SELECT creates a stream where the ith token is either the ith token of stream1 or the ith token of stream2 depending on the boolean value of the ith token of stream3. SUBSTM is used to remove tokens from the "front" or "rear" of its argument stream. Scalar1 tokens are removed from the front of the stream and scalar2 tokens are allowed to pass. The remainder, with the exception of the eos token, are destroyed.

CHAPTER III. PETRI NETS

The Petri net model of concurrent asynchronous activity has gained wide acceptance in the modeling of parallel computing systems [Agerwala 1975, Agerwala 1979, Peterson 1977, Ramamoorthy and Ho 1980, Ramchandani 1974]. This chapter presents a definition of Petri nets along with some relevant developments by other researchers, particularly C. Ramchandani [Ramchandani 1974] in the analysis and characterization of the behavior of a certain subclass of Petri nets.

A Petri net is defined as a bipartite directed graph N = <T,P,A>. This graph contains a set of transitions T, a set of places P, and a set of directed arcs A from places to transitions and transitions to places. A simple example is found in Figure 3.1. Places are denoted by circles and transitions by bars.

The state of a Petri net is determined by its token marking or configuration of tokens in its places. Tokens are denoted as dots within places. When modeling concurrent systems, transitions correspond to abstract operations and tokens represent the availability of certain resources. When modeling programs, these tokens represent data values or control signals. Marked Petri nets may be executed by following certain firing rules. A transition is enabled when all of its input places contain at least one token.

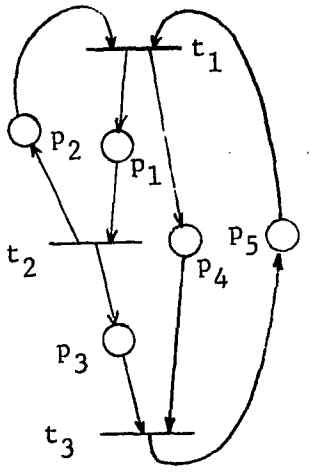
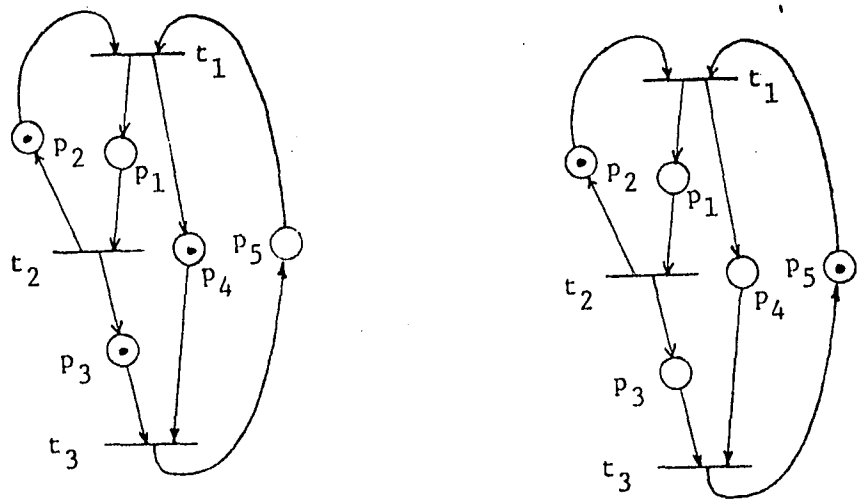


Figure 3.1. Example of a Petri net

The marked Petri net of Figure 3.2(a) has a marking such that transition t_3 is enabled. The firing of a transition removes one token from each input place and adds one token to each output place. When transition t_3 in Figure 3.2(a) fires, it produces the token marking found in Figure 3.2(b) where transition t_1 is now enabled. Other important aspects of concurrent systems may be modeled with Petri nets such as decisions (more than one output arc for a given place) and resource creation from multiple sources (more than one input arc for a given place). For example, the Petri net of Figure 3.3 shows a net with a marking such that transitions t_2 and t_3 are both enabled. When either t_2 or t_3 fires, a token is produced for place p_2, thereby enabling transition t_4.



(a) Marking before firing transition t_3

(b) Marking after firing transition t_3

Figure 3.2. Marked Petri net before and after firing transition

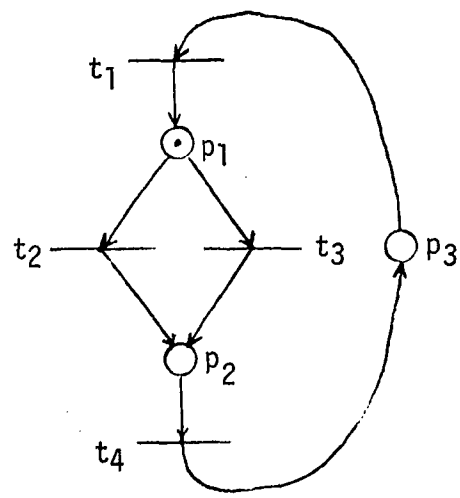


Figure 3.3. Petri net with a decision

Well-structured programs may be modeled by a subclass of Petri nets known as state machine decomposible (SMD) Petri nets. The analysis of this subclass may be totally automated [Ramchandani 1974]. A few formal definitions describe an SMD petri net:

> A Petri net is a state machine iff every transition has only one input and one output place.
>
> A closed subnet $Q_i = \langle T_i, P_i, A_i \rangle$ of a Petri net $\langle T, P, A \rangle$ is a strongly connected Petri net where T_i, P_i, and A_i are subsets of T, P, and A, respectively, and T_i is the set of all input and output transitions of the places of P_i.
>
> A minimal closed subnet is a closed subnet in which no closed subnet may be obtained by deleting any portion of it.
>
> A set of closed subnets $\{Q_i\}$ covers a Petri net N iff $N = \langle UT_i, UP_i, UA_i \rangle$.
>
> A Petri net is SMD iff every minimal closed subnet is a state machine and there exists a set of state machines which covers the net.

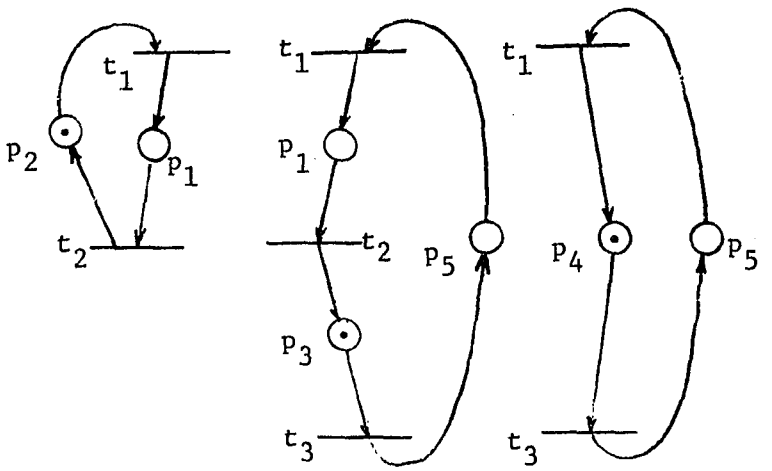


Figure 3.4. State machine decomposition of Petri net of Figure 3.2(a)

The Petri net of Figure 3.2(a) is SMD and is covered by the three state machines found in Figure 3.4.

The relative number of firings of a transition to other transitions in a Petri net is captured in its current assignment [Ramchandani 1974]. The current assignment for a Petri net is made such that every arc carries the current value associated with the transition to which it is connected and the sum of the currents of the input arcs of any place is equal to the sum of the currents of the output arcs of that place. For example, should transition t_2 fire twice as often as transition t_3 then the current assignments could be made for the Petri net of Figure 3.3 of $\phi(t_1) = \phi(t_4) = 3$, $\phi(t_2) = 2$, $\phi(t_3) = 1$ or $\phi(t_1) = \phi(t_4) = 1$, $\phi(t_2) = 2/3$, $\phi(t_3) = 1/3$.

SMD Petri nets have many interesting properties such as a) the number of tokens on any state machine is invariant under transition firings and b) the number of tokens on an SMD Petri net is bounded under transition firings. By assigning a firing time to each transition, a very important characteristic of an SMD Petri net may be computed: its period or average time between firings of its transitions. The period of an SMD Petri net N is computed as the maximum period of all its component state machines:

$$\tau(N) = \max_i [\ \tau(N_i)] \tag{3.1}$$

The period of a state machine N_i is described by the equation

$$\tau(N_i) = \Sigma_{t_j \varepsilon N_i}\ t(t_j) \cdot \phi(t_j) / \Sigma_{p_k \varepsilon N_i}\ M(p_k) \tag{3.2}$$

where $t(t_j)$ is the firing time of transition t_j in N_i and $M(p_k)$ is the number of tokens in place p_k in N_i. Since there exist an infinite set of current assignments for a given Petri net (infinitely many ways of specifying a relative frequency), the current assignment will be chosen such that the transition(s) t_j with the largest relative current will have a current assignment of one: $\phi(t_j) = 1$. That is, all current assignments are less than or equal to one. As an example, consider the Petri net of Figure 3.2(a) and its state machine decomposition of Figure 3.4. All current assignments are one since there are no decisions (places with multiple output arcs) in the Petri net.

Assuming unit execution time for all transitions, the period of the original Petri net is computed as

$$\tau(N) = \max [\ \tau(N_1), \tau(N_2), \tau(N_3)]$$
$$= \max [2/1, 3/1, 2/1] = 3$$

The expected time between firings of a transition t_j in the SMD Petri net N is computed as $\tau(N)/\phi(t_j)$. Therefore, all the transitions of the Petri net of Figure 3.2(a) fire on the average every three time steps. The Petri net N of Figure 3.3 is a state machine and assuming unit execution time for all transitions and

$$\phi(t_1) = \phi(t_4) = 1,\ \phi(t_2) = .3,\ \text{and}\ \phi(t_3) = .7$$
$$\tau(N) = [(1 \cdot 1) + (1 \cdot .3) + (1 \cdot .7) + (1 \cdot 1)] / 1$$
$$= 3$$

Therefore, transitions t_1 and t_4 fire on the average once every three time steps, transition t_2 fires on the average one time every ten time steps (3/.3 time steps between firings), and transition t_3 fires on the average seven times every thirty time steps (3/.7 time steps between firings).

CHAPTER IV. RECURSIVE REDUCTION OF NODES

Through various approximations, a well-structured program may be modeled by SMD Petri nets by representing operations as transitions and data values as tokens. This modeling of the execution of a program in a data flow environment involves the characteristics of the underlying architecture and the features of an appropriate language. The Petri net representation of the static program graph along with its token marking reflects both of these considerations.

The technique for timing analysis of a data flow program which is introduced in this work consists of recognizing "nodes" (or subnets) in the program graph and reducing each node to a single transition (with associated attributes) representing the abstract operation. Nodes correspond to some of the abstract operations defined at the high level:

1) base level machine operations,
2) streamed computations,
3) pre-analyzed high level stream operations (e.g., FIRST and REST),
4) while-do constructs,
5) blocks,
6) conditionals, and
7) procedure invocations.

The base level machine operations (e.g., +, append, SQRT, identity) require no reduction and appear as transitions with firing times equal to the execution time of the operations. (In the examples in this presentation, all base level machine operations are assumed to have unit execution time.)

The reduction process associates with each node a set of attributes which characterizes its stand-alone timing behavior and its potential effect on the timing behavior of any enclosing node. These attributes are derived from the analysis of SMD Petri nets obtained from the node. Some of the attributes are required only if the node is embedded in a stream computation and are dependent on the manner the architecture implements streams. High level stream operators are regarded as pre-analyzed nodes with attributes known a priori and are described in more detail in a subsequent chapter. A node is reduced to a transition if it contains no internal nodes which have not been reduced. Since the high level program may have an arbitrary nesting of constructs, any node of the graph may be nested within another node.

The Petri net representation and analysis of a program is approximate due to two constraints:

Constraint 1: The graph represents one of possibly many implementations of the high level program.

Constraint 2: Nodes, as they are reduced to single transitions, are assumed to begin execution only when all operands of all entry transitions (initial operations) are available and all outputs are simultaneously released.

The first constraint represents an area which is outside the scope of this research and, as a consequence, the term program is taken to mean the original graph itself. Imposing the second constraint results in an approximation of the computation time of the enclosing nodes. However, as pointed out later, this constraint may significantly reduce the time for the analysis.

Performance Analysis of Program Execution

This section presents the technique of nodal reductions of high level program constructs in the approximation of a lower bound on the execution time. This technique is applied to the feedback model, similar to a form of the Dennis-Misunas architecture [Dennis and Misunas 1975]. Other architectures are dealt with in Chapter VI. In the feedback model, an operation contains space for each operand. This space is reusable should the operation be reinvoked (as is the case in the body of an iterative construct by values of successive iterations or in the case of the processing of successive values of a stream within a streamed computation). The feedback model therefore insures

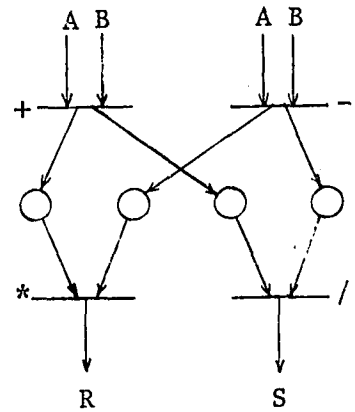
that the output arcs (operand space of successive operations) are clear before an operation may fire. This is accomplished by the use of feedback signals for the purpose of preventing overwrite of data values. For simplicity, feedback arcs will generally be omitted except where needed for analysis. Other architectural proposals allow multiple tokens on each arc by tagging each token through a naming scheme. The notation used in this work is described as it is developed. The reader is referred to Appendix C for an overall summary.

Blocks (scalar)

As has been mentioned earlier, base level operations are modeled by single transitions in a Petri net. The block found in Figure 2.1 (repeated in Figure 4.1(a)) is modeled by the Petri net of Figure 4.1(b). The graphical representation of this construct shows clearly the independence of the + and - operations, and the * and / operations. Given the availability of the values of the identifiers A and B, the + and - operations may be performed simultaneously at time step one producing values for X and Y. The operations * and / at time step two produce the output results of the block: values for R and S. This block therefore requires two time steps to compute its output values given its input values. In general, the maximum path length from any entry (initial) transition to any exit (terminal) transition of

```
BEGIN
REAL X,Y,R,S;
X,Y = A + B, A - B;
R,S = X * Y, X / Y;
END (R,S)
```

(a) Sample Block



(b) Petri net representation

Figure 4.1. Sample block and its representation

the path--weighted by the execution (firing) time of each transition--specifies the nodal firing time of the construct.

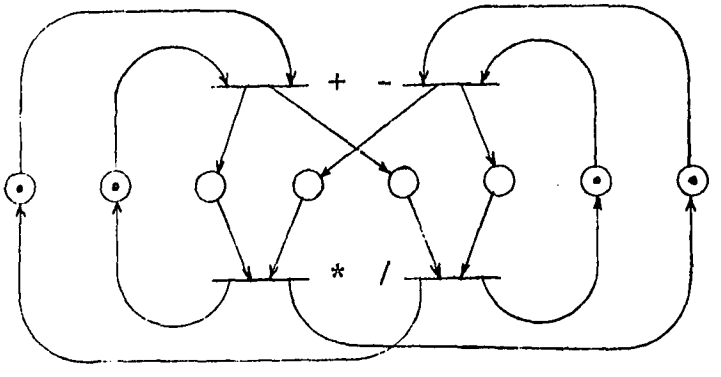
This maximal path length may be computed equivalently as the period of an SMD Petri net Ns_1 constructed from the Petri net N. Ns_1 is the Petri net N with the addition of an acknowledge place initialized with one token from each exit transition to each entry transition. The Petri net Ns_1 constructed from the Petri net N of Figure 4.1(b) is shown in Figure 4.2 along with its state machine decomposition. The period of Ns_1 and the nodal firing time of N is therefore computed using Equations (3.1) and (3.2) (where

all currents are one) as

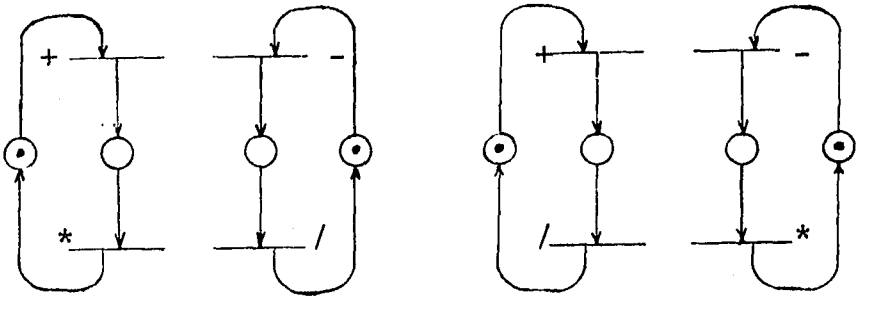
$$\tau(Ns_1) = t_1(N)$$

$$= \max[\ 2/1,\ 2/1,\ 2/1,\ 2/1\ ] = 2$$

Therefore, the block of Figure 4.1 is reduced to a single transition with a nodal firing time $t_1(N) = 2$. This general procedure follows for subsequent constructs with appropriate adjustments.



(a) Petri net Ns_1

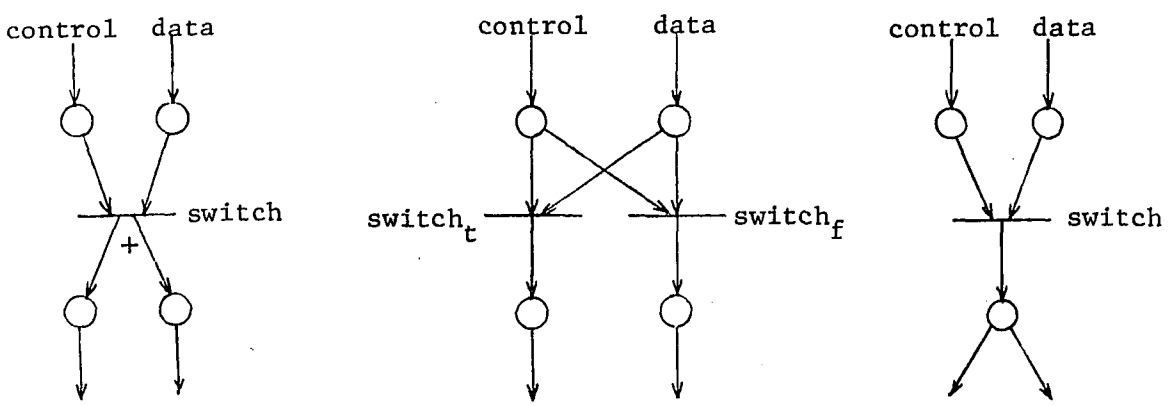


(b) State machine decomposition

Figure 4.2. Petri net Ns_1 and its SMD

Conditionals (scalar)

The feedback model defines two base level operations that are used in the routing of data values to one of two destinations and accepting a data value from one of two sources based upon the logical value of a control signal. The switch operation, Figure 4.3, is modeled directly using an exclusive-or output of an extended Petri net model [Agerwala 1975] in Figure 4.3(a). This operation takes a control signal and a data value and routes the data value to one of two destinations. This operation could be modeled using the ordinary Petri net model as found in Figure 4.3(b) by splitting the switch operation. It is sufficient, however, to model it as found in Figure 4.3(c) for the purposes of this research (except where noted). The merge



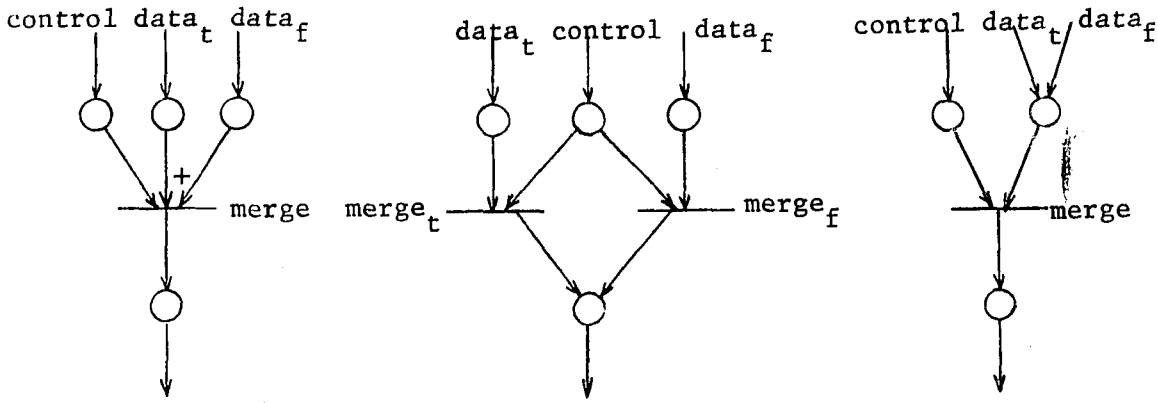
(a) Exclusive-or output

(b) Splitting operation

(c) Ordinary model

Figure 4.3. Switch operation

operation accepts a data value from one of two sources depending on the logical value of its control signal. This operation, likewise, has several representations, as found in Figure 4.4. The model of Figure 4.4(c) is used in this research (except where noted).



(a) Exclusive-or input

(b) Splitting operation

(c) Ordinary model

Figure 4.4. Merge operation

The conditional construct is generally represented as found in Figure 4.5 where the details of the condition have been abstracted out and the then and else bodies have been previously reduced. Due to Constraint 2, it may be assumed that all switches fire simultaneously and all merges fire simultaneously and the representation may be reduced by using generic switch and merge operations to that as found in Figure 4.6(a). This representation may be further reduced by the elimination of unnecessary arcs as found in

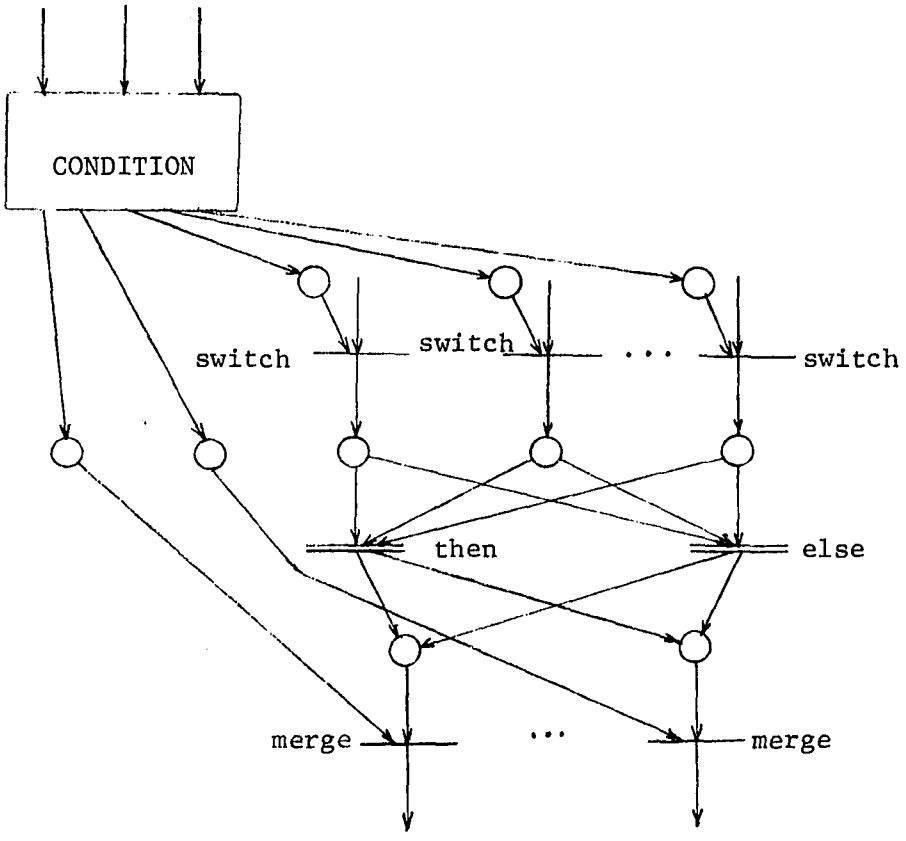


Figure 4.5. Representation of conditional

Figure 4.6(b). This is justified by assuming that the control signal generated by the evaluation of the condition contains the data values of the in-set of the body of the condition, and that this control signal is passed through the then or else body to the merge operation. Realistically, this argument is justified by noting that

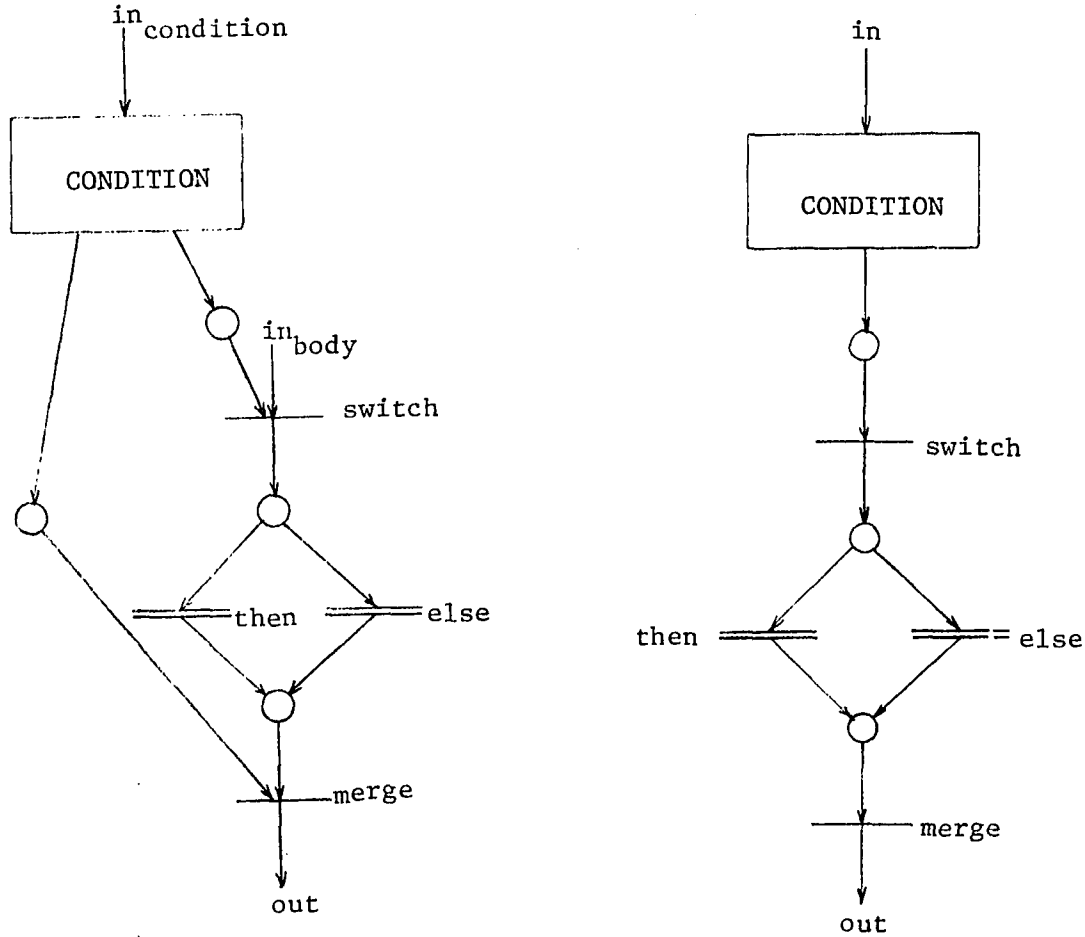
1) by Constraint 2, the evaluation of the condition is not held up by the availability of in_{body} and

2) for every state machine involving the condition and merge there exists a state machine additionally involving the switch, then, and else bodies that has a larger period.

Therefore, the period of any Ns-type Petri net constructed from the Petri net of Figure 4.6(b) would not differ from the period of any Ns-type Petri net constructed from the Petri net of Figure 4.6(a). The Ns-type Petri net constructed from the Petri net of Figure 4.6(b) appears in Figure 4.7. Depending upon the number of entry transitions of the conditional, there might be more than one acknowledge arc from the exit (merge) transition. Due to the occurrence of a decision, not all current assignments are one. The current assignment of the then body is denoted as α and consequently the current assignment of the else body is $1 - \alpha$ (all other transitions have a current assignment of one). As an example, the conditional construct of Figure 4.8 is reduced via the construction of the Ns-type net of Figure 4.9. Since $t_1(\text{then}) = 1$ and $t_1(\text{else}) = 2$, the period of the SMD Petri net of Figure 4.9(a) is computed as:

$$
\begin{aligned}
t_1(N) &= \tau(Ns_1) \\
&= \max \{ [4 + \alpha \cdot t_1(\text{then}) + (1 - \alpha) \cdot t_1(\text{else})] / 1, \\
&\qquad [5 + \alpha \cdot t_1(\text{then}) + (1 - \alpha) \cdot t_1(\text{else})] / 1 \} \\
&= 5 + \alpha \cdot t_1(\text{then}) + (1 - \alpha) \cdot t_1(\text{else}) \\
&= 5 + \alpha + 2(1 - \alpha) = 7 - \alpha
\end{aligned}
$$

α is a parameter to the analysis representing the expected probability of choosing the then side and, hence, $t_1(N)$ represents the expected execution time of the conditional.



(a) Generic switch and merge operations

(b) Elimination of unnecessary arcs

Figure 4.6. Reduced conditional representation

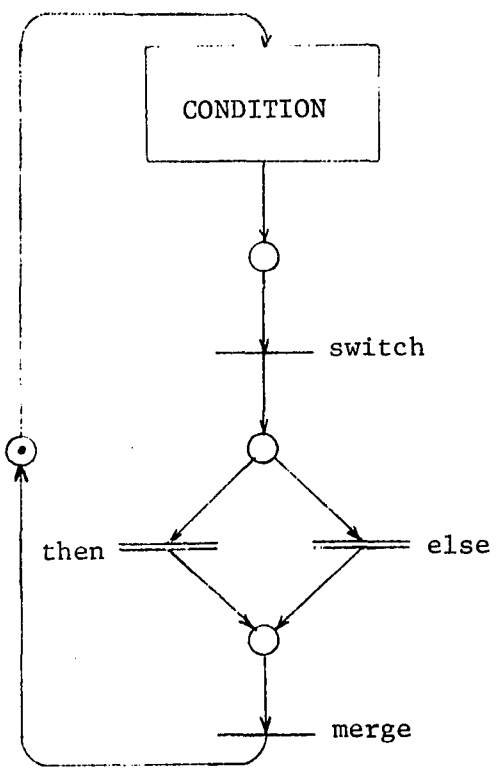
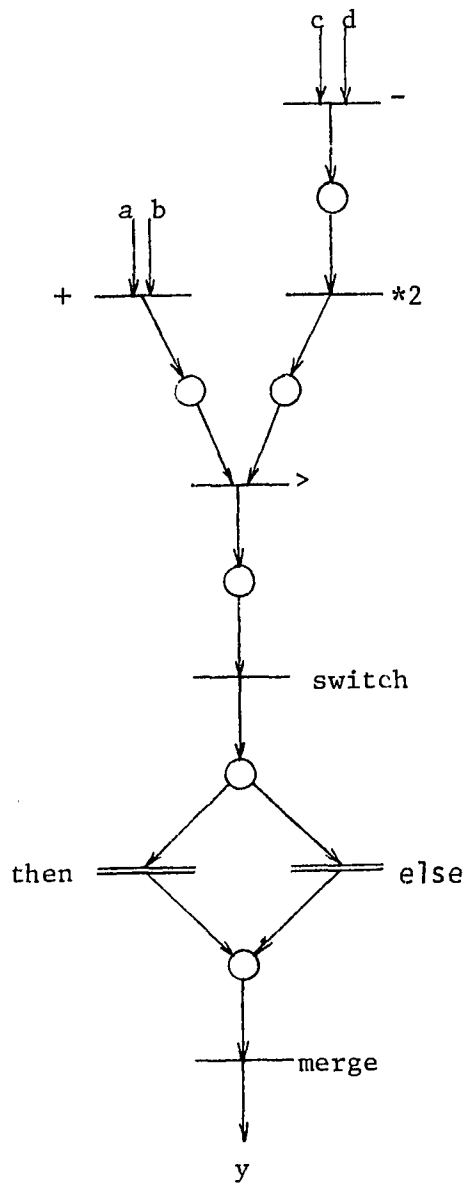


Figure 4.7. Ns-type Petri net of a conditional

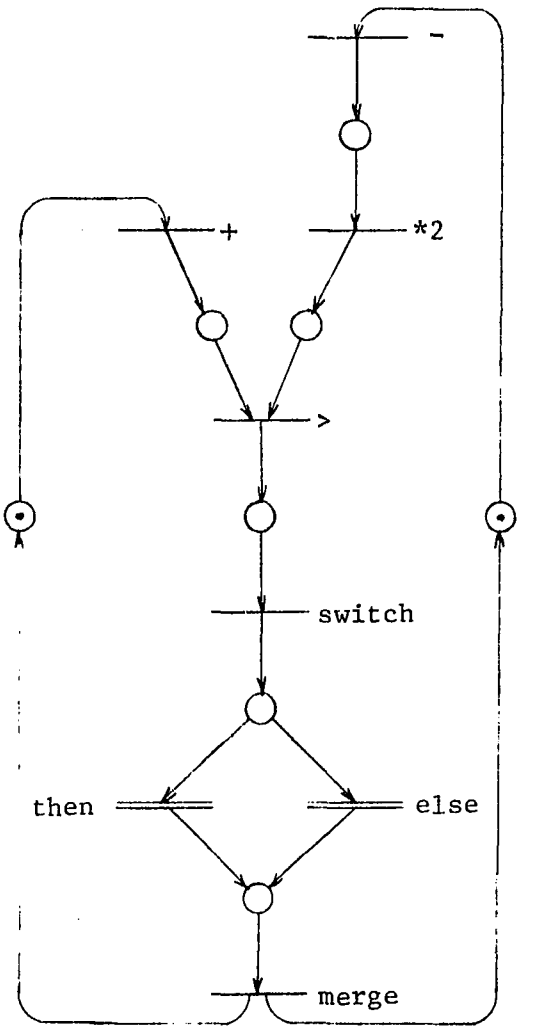
```
IF (a + b) > (c - d) * 2
  THEN BEGIN
       INTEGER x
       x = a + b
       END (x)
  ELSE BEGIN
       INTEGER y
       y = (c - d) * 2
       END (y)
```

(a) High level code



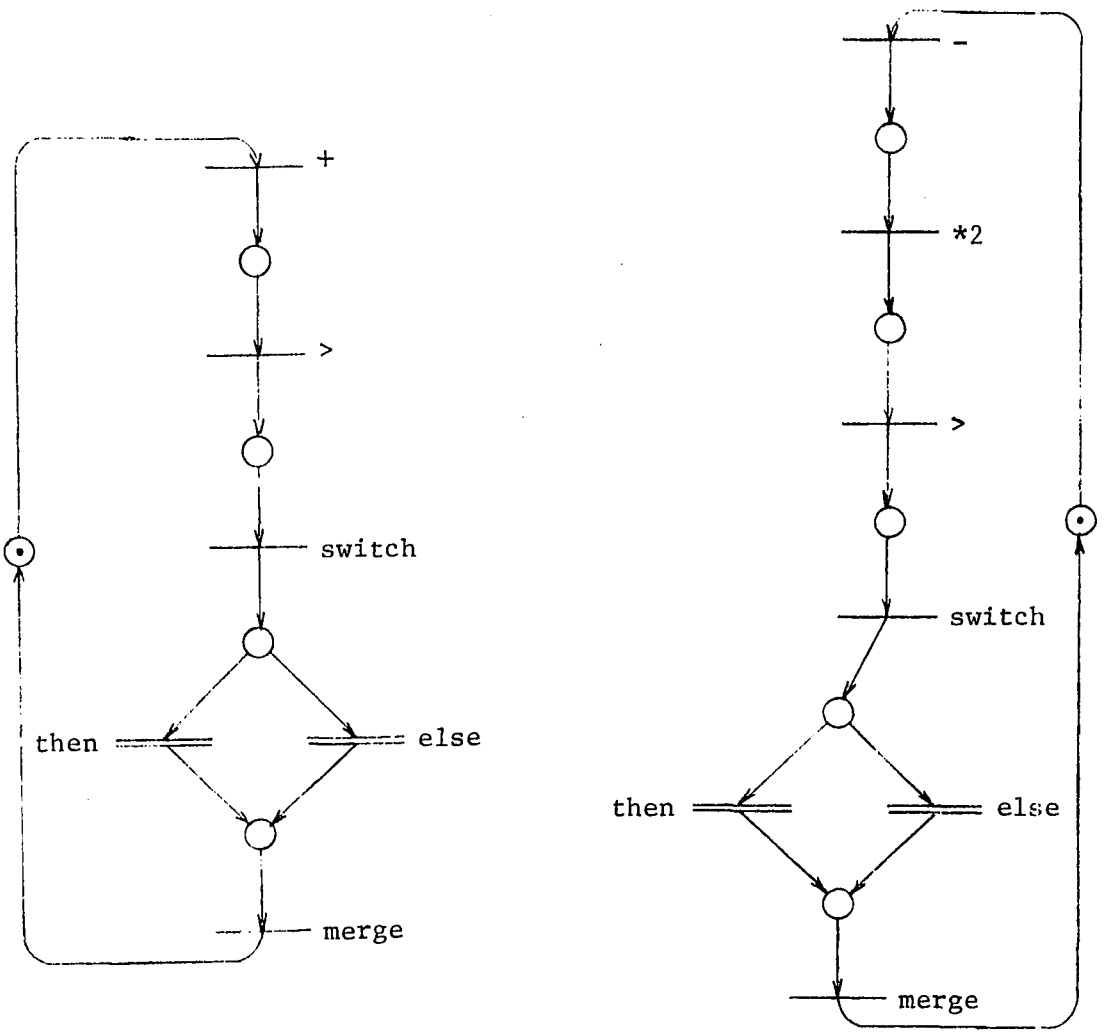
(b) Petri net representation N

Figure 4.8. Example conditional construct



(a) Ns-type Petri net

Figure 4.9. Ns-type Petri net of Figure 4.8(b)



(b) State machine decomposition

Figure 4.9. Continued

While-do

The while-do is assumed to be implemented as an iterative computation, reusing the program graph in each iteration. This constraint of reusing nodes in the graph is relaxed for the architecture of Chapter VI. The analysis of a recursive implementation is handled in a subsequent section on recursive procedures. In the feedback model, feedback signals prevent the simultaneous activity of more than one iteration. Therefore, the total execution time of a while-do node N is computed as

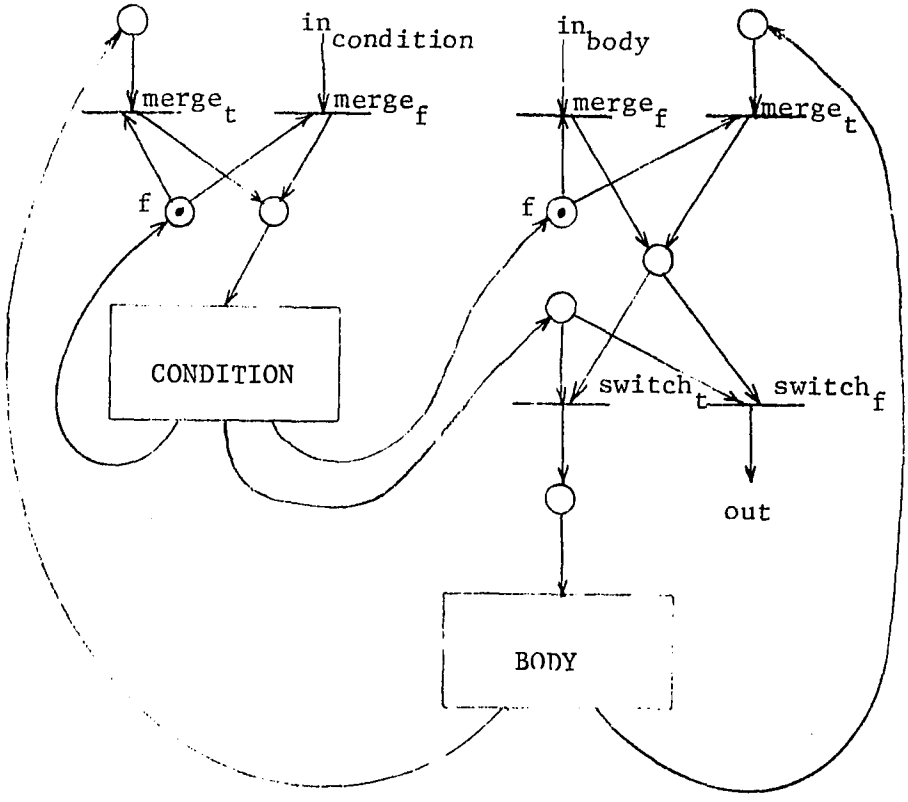
$$t_1(N) = m \cdot \tau(W) + \tau(F) \qquad (4.1)$$

where the parameter m is an externally supplied value for the number of loop iterations. W is the Petri net constructed from N to model the first m iterations and F is the Petri net constructed to model the final test.

The general template of a while-do node is represented by the Petri net N of Figure 4.10(a) with additional semantics (involving the boolean values) implied to properly describe the overall firing patterns. This representation splits the switch and merge operations as previously described and illustrated in Figures 4.3(b) and 4.4(b), respectively. The details of the condition and body have been abstracted out. These implied semantics are not necessary in the representation of W and F, appearing in Figures 4.10(b) and 4.10(c), since each uses one of the

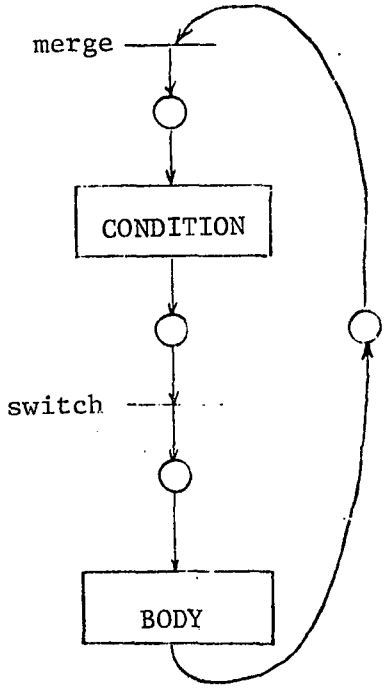
split switch and split merge nodes. Those statements appearing in the initial clause of a while-do are assumed to be contained in the enclosing node and are not reduced here.

As an example, consider the while-do found in Figure 2.2 (reproduced in Figure 4.11(a)). The Petri nets W and F modeling this construct are found in Figures 4.11(b) and 4.11(c), respectively. The periods of the Petri nets W and

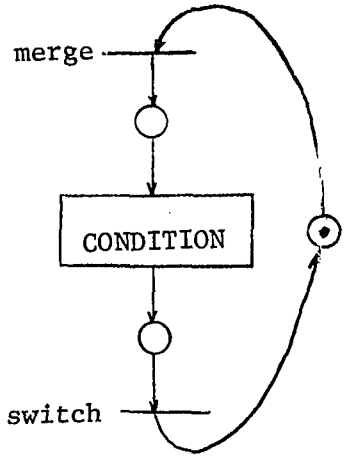


(a) Petri net representation of While-do node N

Figure 4.1Ø. Petri net representations of while-do node



(b) Petri net representation W of iteration of while-do node



(c) Petri net representation F of final test of a while-do

Figure 4.10. Continued

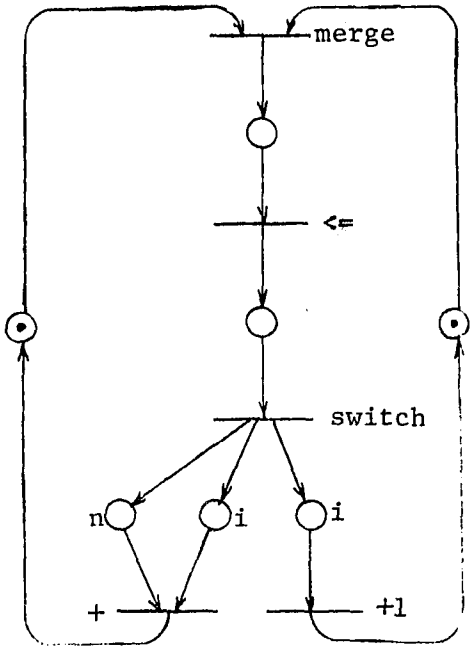
F are computed as $\tau(W) = 4$ and $\tau(F) = 3$, hence, Equation (4.1) becomes $t_1(N) = 4(m) + 3$ where the parameter m would be given the value 10.

Procedure Invocations (scalar)

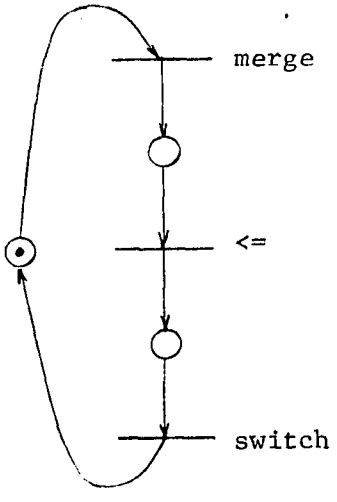
Procedure calls are treated as though a copy of the procedure was dynamically inserted into the program graph at the time of the procedure call. This is depicted in Figure

```
WHILE i <= 10
   INITIAL i=1, n=0 DO
   INTEGER i,n;
   new n = n + i;
   new i = i + 1
   END (n)
```

(a) While-do N



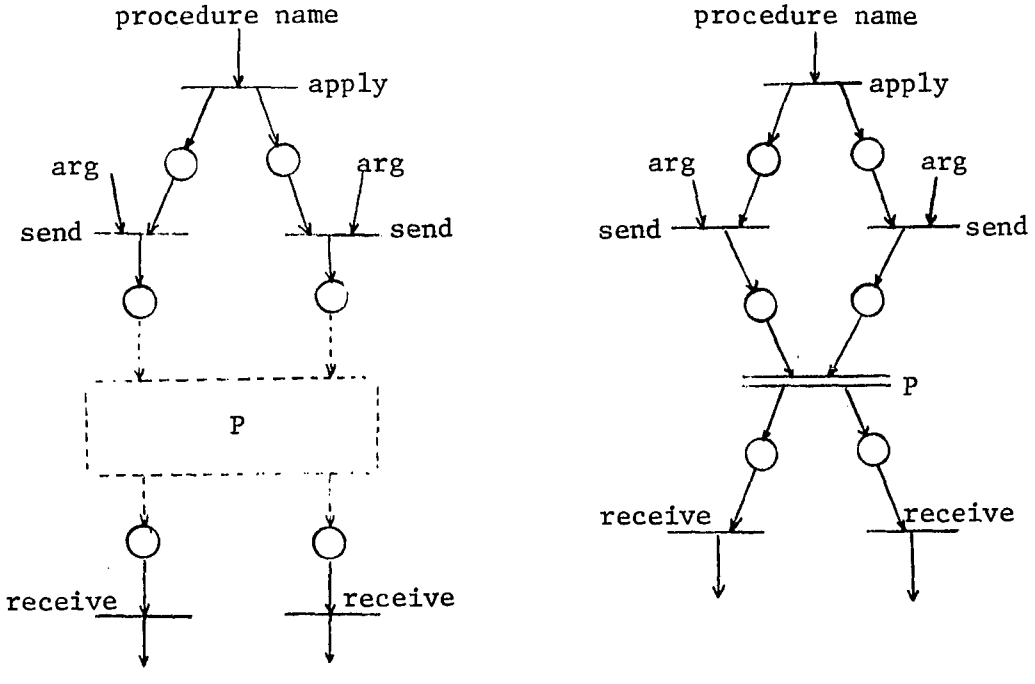
(b) Petri net W



(c) Petri net F

Figure 4.11. Sample while-do

4.12. A program is treated as a procedure and therefore its nodal firing time characterizes the total execution time. The procedure implementation described here is one in which each parameter has a separate input or output port. This is accomplished via the special operations "send" and "receive" found both in the invoking construct and in the invoked procedure. Other implementations could be modeled



(a) Dynamic insertion of procedure body

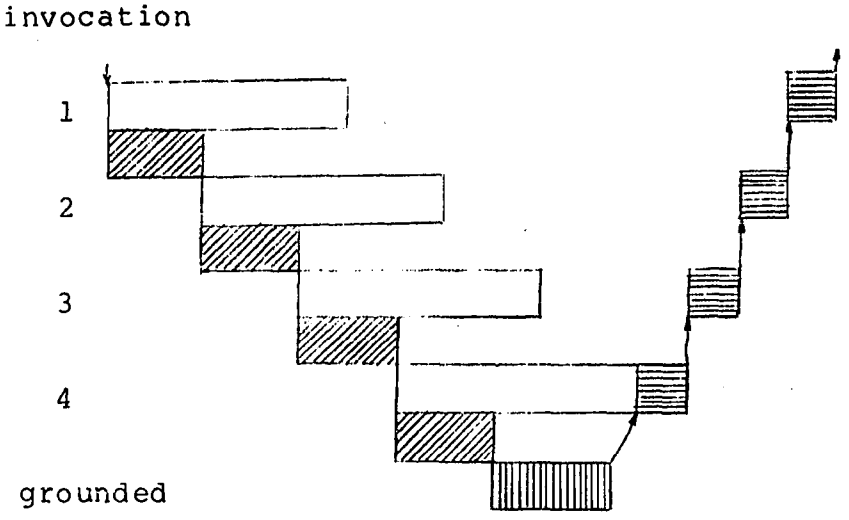
(b) Petri net after nodal reduction

Figure 4.12. Procedure invocation

with appropriate adjustments. The reduction of a non-recursive procedure N occurs in a straightforward fashion with the construction and analysis of the corresponding Ns-type Petri net obtained from N by the addition of acknowledge places initialized with one token each from all exit nodes to all entry nodes.

Recursive procedures have the advantage that when used to implement computations that are conventionally done iteratively, a marked increase in parallelism may result. This is due to the supplying of additional copies of procedure code whereas a single body of an iteration is reused. A timing diagram appears in Figure 4.13 that illustrates the potential overlap of the computation that occurs within different invocations. Recursive procedures necessarily involve conditionals and with appropriate (but perhaps difficult) calculation of α-values and the solving of a recursive timing equation(s), this class of procedures may be analyzed by the standard methodology.

Three examples are presented that compute the sum of the elements of an array. The iterative computation of Figure 4.14(a) is expressed recursively via an initial invocation of add(A,1,n) of the procedure of Figure 4.14(b). Should the computation involving A(i) in each iteration have been more complex, the real advantage of a recursive expression would be more noticeable due to overlap of computations involved in successive invocations. The



time to calculate values produced by the body of this invocation

time to initiate recursive call

time of grounded invocation

time to coalesce and return values produced by this invocation and a recursive call

Figure 4.13. Timing diagram of execution of a recursive procedure producing single token results

recursive procedure of Figure 4.14(b) is analyzed through the nodal reductions of Figures 4.15(a) through 4.15(d). Assuming unit time for base level operations, the computation of t_1(add) is accomplished through construction of appropriate Ns-type Petri nets and their SMD. The resulting computation is

```
WHILE i <= n
   INITIAL i = 1,
      add = Ø DO
   INTEGER i, add;
   new add = add + a(i)
   new i = i + 1
   END (add)
```

(a) Iterative computation

```
PROCEDURE add (INTEGER ARRAY a,
      INTEGER i,n)
      RETURNS (INTEGER s)
 s = IF i = n
      THEN BEGIN
           INTEGER x;
           x = a(n)
           END (x)
       ELSE BEGIN
           INTEGER y;
           y = a(i) +
               add(a,i+1,n);
           END (y)
   END
```

(b) Recursive computation

Figure 4.14. Linear accumulations of the sum of the elements of an array

$$t_1(\text{add}) = t(\text{receive}) + t_1(\text{if}) + t(\text{send})$$

$$= 2 + t_1(\text{if})$$

$$t_1(\text{if}) = t(=) + t(\text{switch}) + \alpha \cdot t_1(\text{then}) + (1 - \alpha) \cdot t_1(\text{else}) + t(\text{merge})$$

$$= 3 + \alpha \cdot t_1(\text{then}) + (1 - \alpha) \cdot t_1(\text{else})$$

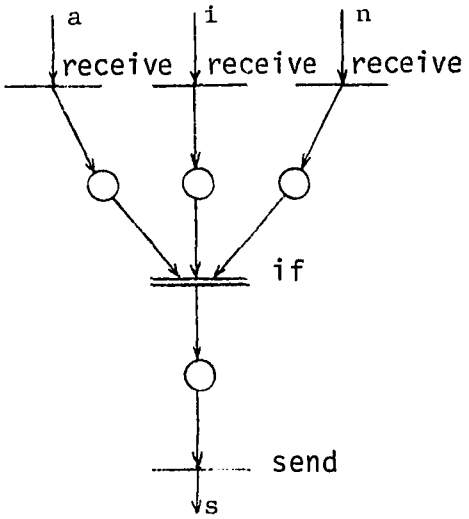
$$t_1(\text{then}) = t(\text{select}) = 1$$

$$t_1(\text{else}) = \max [t(\text{select}) + t(+), t(\text{send}) + t_1(\text{add}) + t(\text{receive}) + t(+), t(+) + t(\text{send}) + t_1(\text{add}) + t(\text{receive}) + t(+), t(\text{send}) + t_1(\text{add}) + t(\text{receive}) + t(+)]$$

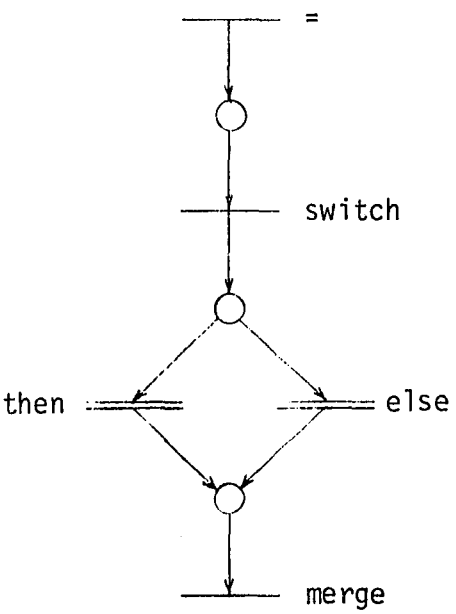
$$= t(+) + t(\text{send}) + t_1(\text{add}) + t(\text{receive}) + t(+)$$

$$= 4 + t_1(\text{add})$$

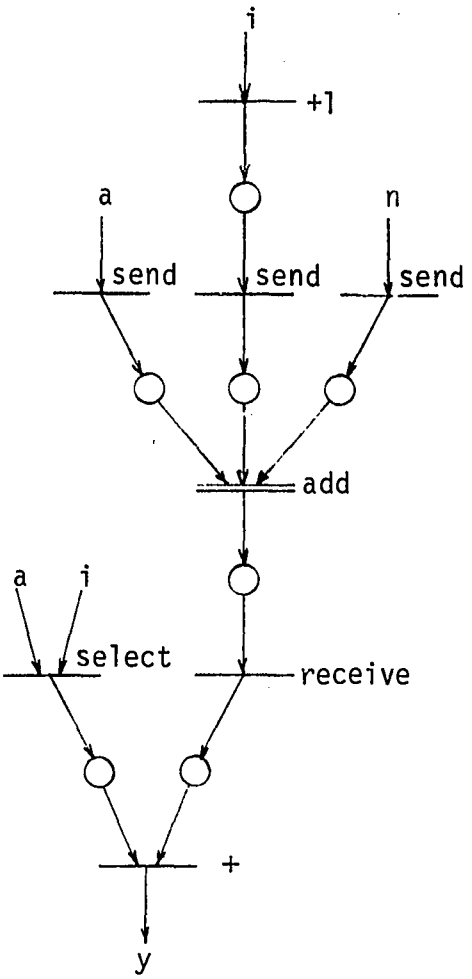
Therefore,



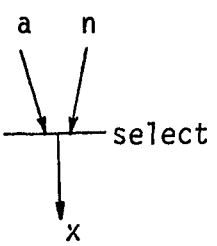
(a) Procedure



(b) If



(c) Else



(d) Then

Figure 4.15. Nodal reductions of the procedure add

$$t_1(add) = 2 + \{3 + \alpha[1] + (1 - \alpha)[4 + t_1(add)]\}$$
$$= 5 + \alpha + 4(1 - \alpha) + (1 - \alpha)\cdot t_1(add)$$
$$= 9 - 3(\alpha) + (1 - \alpha)\cdot t_1(add)$$
$$= 9/\alpha - 3 \qquad (4.2)$$

The recursive procedure of Figure 4.16 computes the sum in $O(\log_2 x)$ time for an array A of x elements. This procedure is analyzed through the nodal reductions found in Figure 4.17(a) through 4.17(d). The computation of t (log) proceeds through the following steps with the appropriate construction of the Ns-type Petri nets:

$$t_1(log) = t(receive) + t_1(if) + t(send)$$
$$= 2 + t_1(if)$$

```
PROCEDURE log (INTEGER ARRAY a, INTEGER l,h)
      RETURNS (INTEGER n)
   n = IF l = h
         THEN BEGIN
              INTEGER x;
              x = a(h)
              END (x)
         ELSE BEGIN
              INTEGER m,y;
              m = (l + h)/2;
              y = log(a,l,m) + log(a,m+1,h)
              END (y)
   END
```

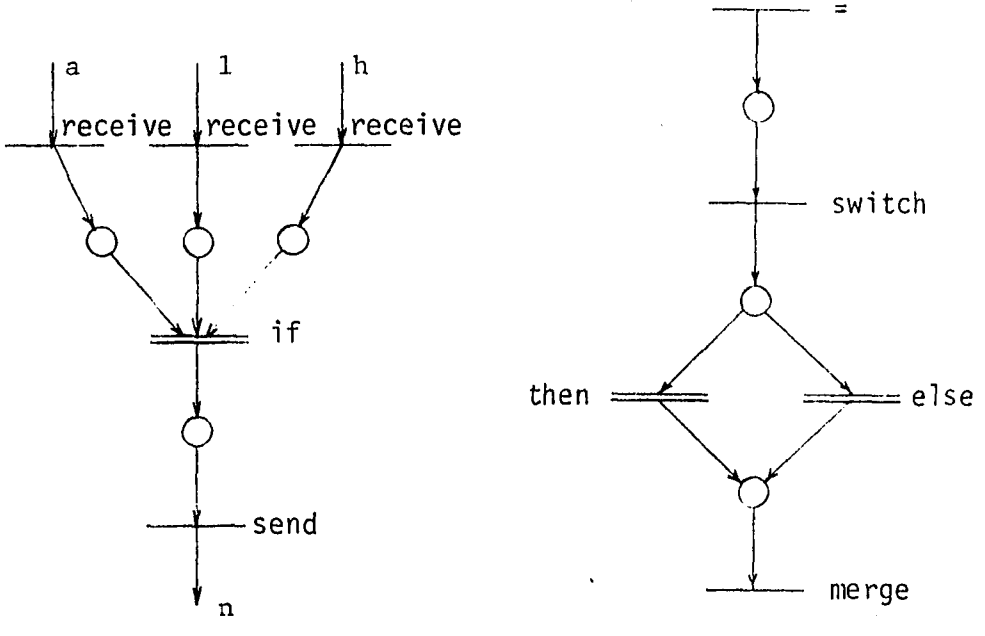
Figure 4.16. Procedure to compute sum of the x elements of an array in $O(\log_2 x)$ time

$$t_1(\text{if}) = t(=) + t(\text{switch}) + \alpha \cdot t_1(\text{then}) + (1 - \alpha) \cdot t_1(\text{else}) + t(\text{merge})$$

$$= 3 + \alpha \cdot t_1(\text{then}) + t_1(\text{else}) \cdot (1 - \alpha)$$

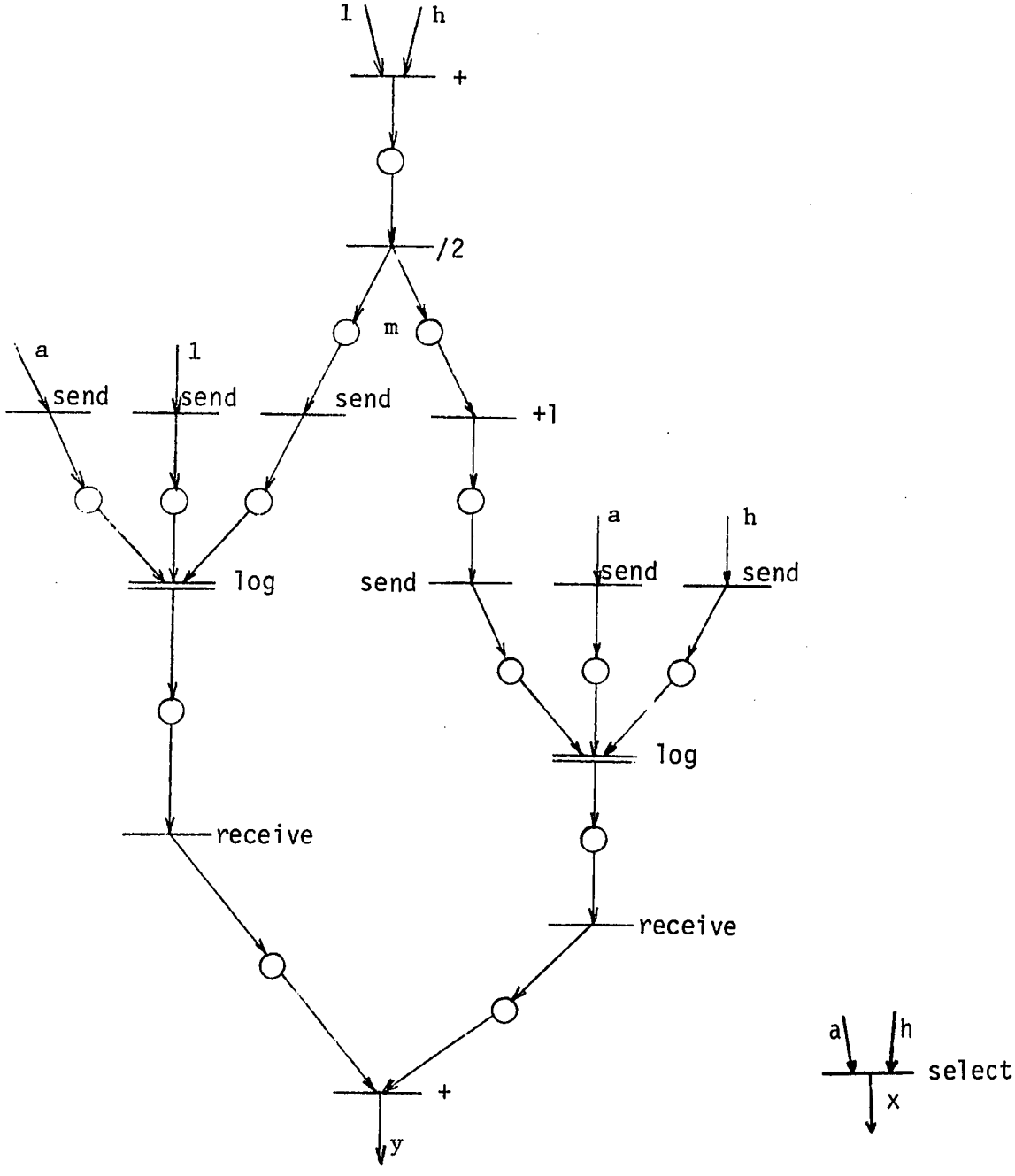
$$t_1(\text{then}) = t(\text{select}) = 1$$

The Ns-type Petri net for the else body of Figure 4.17(c) decomposes into six state machines. Four of these have the same period that is dominated by the remaining two state machines' periods:



(a) Procedure (b) If

Figure 4.17. Nodal reductions of the procedure log



(c) Else (d) Then

Figure 4.17. Continued

$$t_1(\text{else}) = \max [t(+) + t(/) + t(\text{send}) + t_1(\log) + t(\text{receive}) + t(+),\ t(+) + t(/) + t(+) + t(\text{send}) + t_1(\log) + t(\text{receive}) + t(+)]$$

$$= t(+) + t(/) + t(+) + t(\text{send}) + t_1(\log) + t(\text{receive}) + t(+)$$

$$= 6 + t_1(\log)$$

Therefore,

$$t_1(\log) = 2 + \{3 + \alpha[1] + (1 - \alpha)[6 + t_1(\log)]\}$$

$$= 5 + \alpha + 6(1 - \alpha) + (1 - \alpha)\cdot t_1(\log)$$

$$= 11 - 5(\alpha) + (1 - \alpha)\cdot t_1(\log)$$

$$= 11/\alpha - 5 \qquad (4.3)$$

The recursive procedure of Figure 4.18 accumulates the sum in a different manner (though not efficiently). This procedure is analyzed through the nodal reductions of Figures 4.19(a) through 4.19(d). The computation of $t_1(\text{sum})$ is accomplished through the appropriate construction of the Ns-type Petri nets and the following calculations:

$$t_1(\text{sum}) = t(\text{receive}) + t_1(\text{if}) + t(\text{send})$$

$$= 2 + t_1(\text{if})$$

$$t_1(\text{if}) = t(=) + t(\text{switch}) + \alpha\cdot t_1(\text{then}) + (1 - \alpha)\cdot t_1(\text{else}) + t(\text{merge})$$

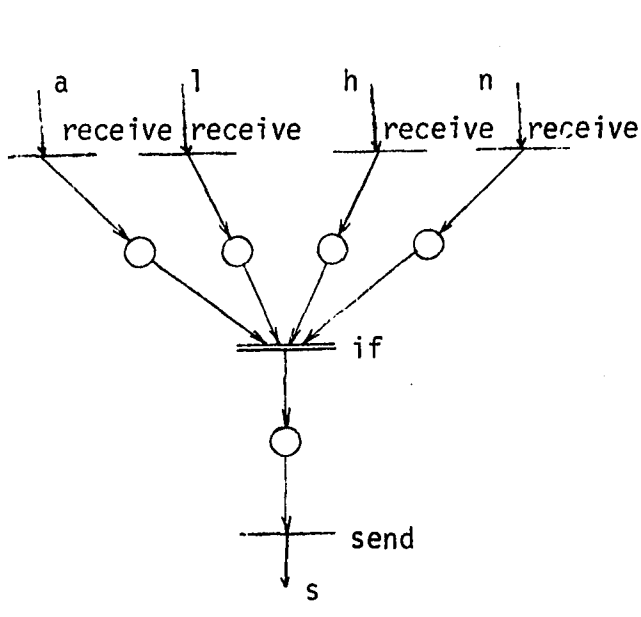
$$= 3 + \alpha\cdot t_1(\text{then}) + (1 - \alpha)\cdot t_1(\text{else})$$

$$t_1(\text{then}) = t(\text{select}) + t(+) = 2$$

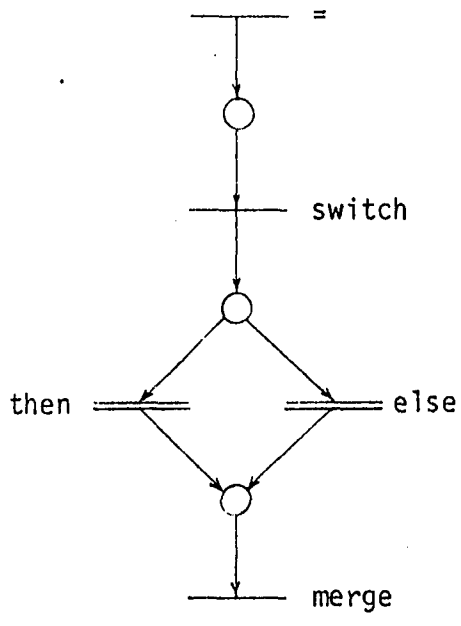
There are seven state machines in the decomposition of the

```
PROCEDURE sum (INTEGER ARRAY a, INTEGER l,h,n)
     RETURNS (INTEGER s)
  s = IF l = h
        THEN BEGIN
             INTEGER x;
             x = a(l) + n
             END (x)
        ELSE BEGIN
             INTEGER m,p,y;
             m = (l + h)/2;
             p = sum(a,l,m,n);
             y = sum(a,m+1,h,p)
             END (y)
  END
```

Figure 4.18. Procedure computing sum of elements of an array with data dependencies between multiple recursive invocations (initial call on sum has actual parameter n = 0)

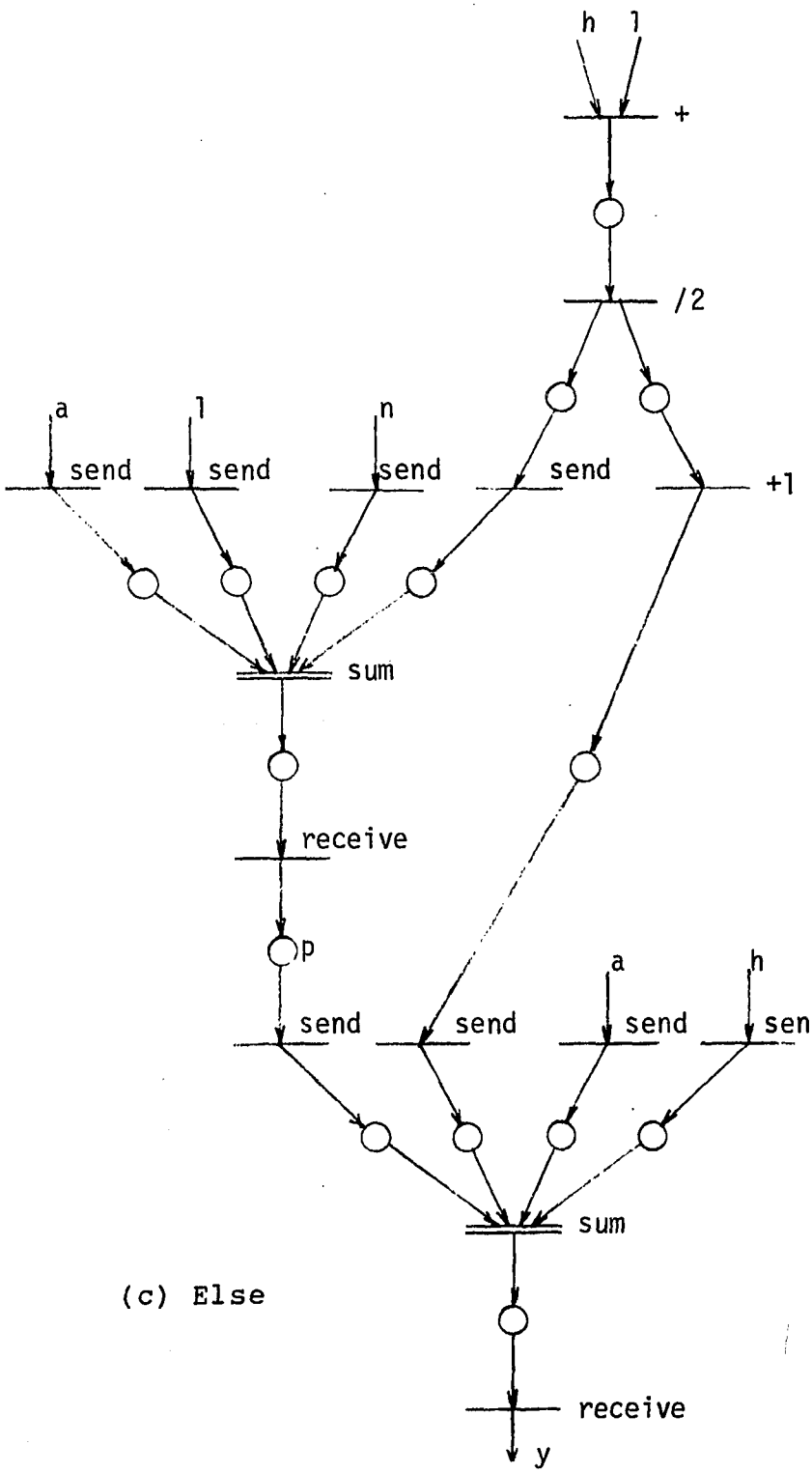


(a) Procedure

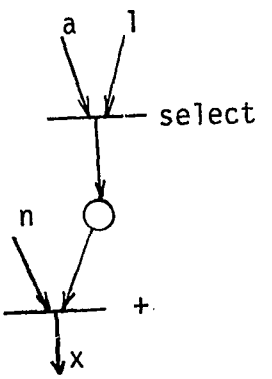


(b) If

Figure 4.19. Petri net representation of the procedure of Figure 4.18



(c) Else



(d) Then

Figure 4.19. Continued

Ns-type Petri net constructed from the else node of Figure 4.19(c). The two major ones appear in the following computation:

$$\begin{aligned} t_1(\text{else}) &= \max[t(+) + t(/) + t(\text{send}) + t_1(\text{sum}) + \\ &\qquad t(\text{receive}) + t(\text{send}) + t_1(\text{sum}) + \\ &\qquad t(\text{receive}),\ t(+) + t(/) + t(+) + \\ &\qquad t(\text{send}) + t_1(\text{sum}) + t(\text{receive})] \\ &= \max[2 \cdot t_1(\text{sum}) + 6,\ t_1(\text{sum}) + 5] \\ &= 2 \cdot t_1(\text{sum}) + 6 \end{aligned}$$

Therefore,

$$\begin{aligned} t_1(\text{sum}) &= 2 + \{3 + \alpha[2] + (1 - \alpha)[2 \cdot t_1(\text{sum}) + 6]\} \\ &= 5 + 2\alpha + 2 \cdot (1 - \alpha) \cdot t_1(\text{sum}) + (1 - \alpha)6 \\ &= [5 + 2\alpha + 6(1 - \alpha)] \ / \ [1 - 2(1 - \alpha)] \\ &= (11 - 4\alpha) \ / \ (2\alpha - 1) \end{aligned} \qquad (4.4)$$

The calculation of appropriate α-values is very crucial to this method. This may not always be easily accomplished since it is dependent upon the parallelism exhibited at the procedure level which may not be obvious. The behavior of the previous three examples is summarized by the diagrams of Figure 4.2Ø for an array with four elements. The blocks labeled "n" indicate those invocations that are non-grounded and those labeled "t" are the terminal or grounded invocations. Solid lines denote a recursive call, and in Figure 4.2Ø(c), dashed lines indicate a data dependency. The non-grounded invocations execute the "else" body and the

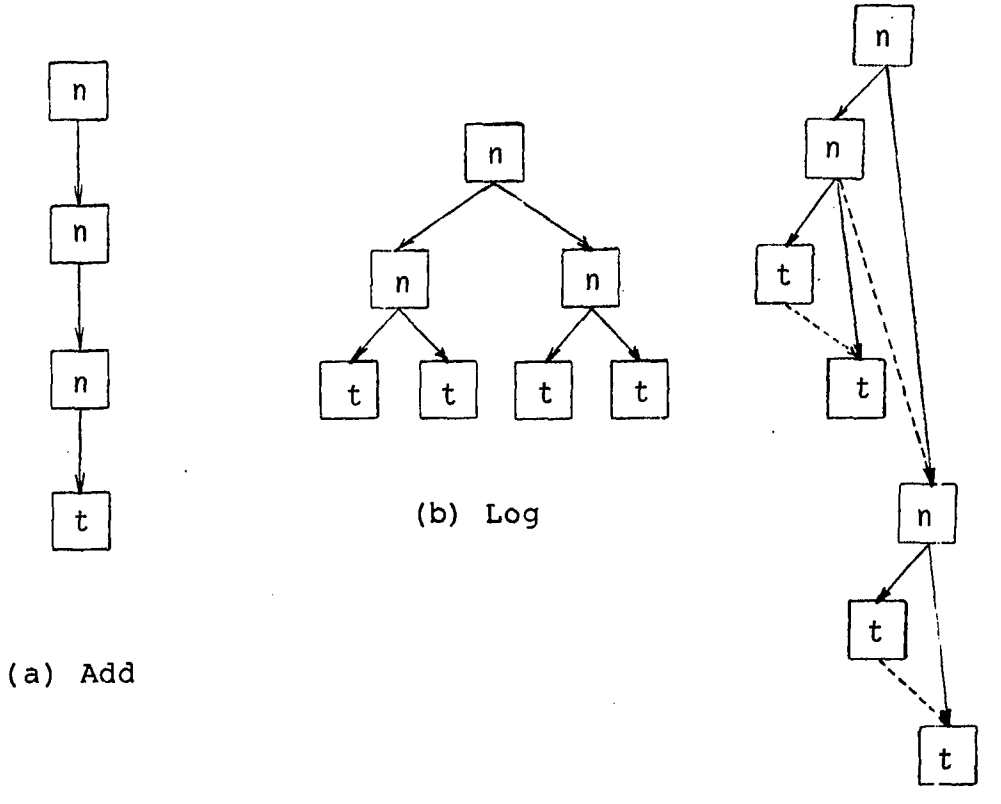


Figure 4.2Ø. Parallelism at the procedure level

grounded invocations execute the "then" body in all three examples. The appropriate calculation of the α-values involves the longest path length through the procedure invocations and (in these examples) is dependent upon the number of elements, n, in the array. For the procedure "add", α is computed as 1/n. For the procedure "log", α is computed as $\lceil \log n + 1 \rceil^{-1}$. And for the procedure "sum", α

is computed as n/(2n-1). As a consequence, should n = 15, Equations (4.2) through (4.4) become:

4.2) $t_1(\text{add}) = 9/\alpha - 3$ where $\alpha = 1/n = 1/15$

$= 9(15) - 3 = 132$

4.3) $t_1(\text{log}) = 11/\alpha - 5$ where $\alpha = 1/\lceil \log n + 1 \rceil = 1/5$

$= 11(5) - 5 = 50$

4.4) $t_1(\text{sum}) = (11 - 4\alpha)/(2\alpha - 1)$

where $\alpha = n/(2n - 1) = 15/29$

$= [11(29) - 4(15)] / [2(15) - 29]$

$= 259$

In the calculation of Equations (4.2) through (4.4), determination of the dominant term of the maximum function was possible. This may not always be the case as is illustrated in the following example found in Figure 4.21. The conditions have been abstracted out and the timing equations take the general form of:

$$t_1(P) = c_1 + (\alpha_1)[c_2 + t_1(Q)] + (1 - \alpha_1)[c_3]$$
$$= k_1 + \alpha_1 \cdot t_1(Q)$$
$$t_1(Q) = c_4 + (\alpha_2)[c_5 + t_1(R)] + (1 - \alpha_2)[c_6]$$
$$= k_2 + \alpha_2 \cdot t_1(R)$$
$$t_1(R) = c_7 + \max\{t_1(Q), c_8 + (\alpha_3)[c_9 + t_1(R)] + (1 - \alpha_3)[c_{10}]\}$$
$$= c_7 + \max[t_1(Q), k_3 + \alpha_3 \cdot t_1(R)]$$

Solving for $t_1(R)$,

```
PROCEDURE P (REAL m) RETURNS (REAL n)
     n = IF β1 THEN BEGIN
                    REAL x;
                    x = Q(m)
                    END (x)
               ELSE BEGIN
                    REAL y;
                    y = m
                    END (y)
     END

PROCEDURE Q (REAL m) RETURNS (REAL n)
     REAL s;
     s = IF β2 THEN BEGIN
                    REAL x;
                    x = R(m)
                    END (x)
               ELSE BEGIN
                    REAL y;
                    y = m
                    END (y);
     n = P(s - 1)
     END

PROCEDURE R (REAL m) RETURNS (REAL n)
     REAL s,t;
     s = Q(m)
     t = IF β3 THEN BEGIN
                    REAL x
                    x = R(m/2)
                    END (x)
               ELSE BEGIN
                    REAL y
                    y = m
                    END (y);
     n = s + t
     END
```

Figure 4.21. Direct and indirect recursion

$$t_1(R) = c_7 + \max[k_2 + \alpha_2 \cdot t_1(R),\ k_3 + \alpha_3 \cdot t_1(R)]$$

Since this equation is not solvable until α_2 and α_3 are known and since $t_1(P)$ and $t_1(Q)$ are dependent upon $t_1(R)$, the user, as the supplier of α_2 and α_3, must solve these equations. Alternatively, both equations could be generated with the user making the determination as to which resultant equation is appropriate based on whether or not

$$k_2 + \alpha_2\ t_1(R) > k_3 + \alpha_3\ t_1(R).$$

Streamed Computations

A streamed computation is a computation defined over a sequence of values [Dennis and Weng 1979, Morris and Treleaven 1975, Morrison 1978, Weng 1975, Weng 1979]. In the feedback model being described, streams are considered to be sequences of tokens residing on the program graph. If the program graph does not provide enough token space for the elements of the stream, buffering and unbuffering (using arrays) occurs. The analysis of program execution where streams are represented as single tokens (pointers to sets of stored elements) is considered in a subsequent chapter dealing with the stream-oriented model. A streamed computation is identified as a Petri net with specific previously reduced nodes serving as sources and sinks of sequences of values. A stream source produces a stream of values given a set of "scalar" or "single-token" values. A

stream sink consumes a stream of values and produces a set of "scalar" or "single-token" values. Portions of the computation on one set of values may be overlapped with computations on other values, allowing the program graph to function as a pipeline. The total execution time of the streamed computation N is therefore expressed as

$$ts(N) + m \cdot tp(N) \qquad (4.5)$$

where the stream contains m values followed by an eos token. The constant portion of Expression (4.5), ts(N), is similar to the "start-up" or "flush" time of a vector operation. However, ts(N) actually represents the time required to perform the stream computation upon the empty stream (m = Ø). "Start-up" or "flush" time does not adequately describe this term since a vector operation takes as input a vector (or stream) and produces a vector (or stream), whereas a streamed computation takes "scalar" values as inputs and produces "scalar" values as outputs, internally producing and consuming streams. The periodic behavior of the streamed computation is captured in the term tp(N). A simplifying assumption here, of little consequence, is that the delay between the last data token and the eos token, $d_{last,eos}$, is not substantially different from the average delay between successive data tokens.

Sample streamed computation An example of a simple streamed computation appears in Figure 4.22 in high level

```
STREAMED
   REAL STREAM A,B,C,D;
   REAL X;

   A = SCREATE(1, m, 1);
   STR-INPUT B FROM FILE1;

   C = (A + B) * B;
   D = A / C;

   X = SUM(D);

   END (X)
```

Figure 4.22. High level streamed computation

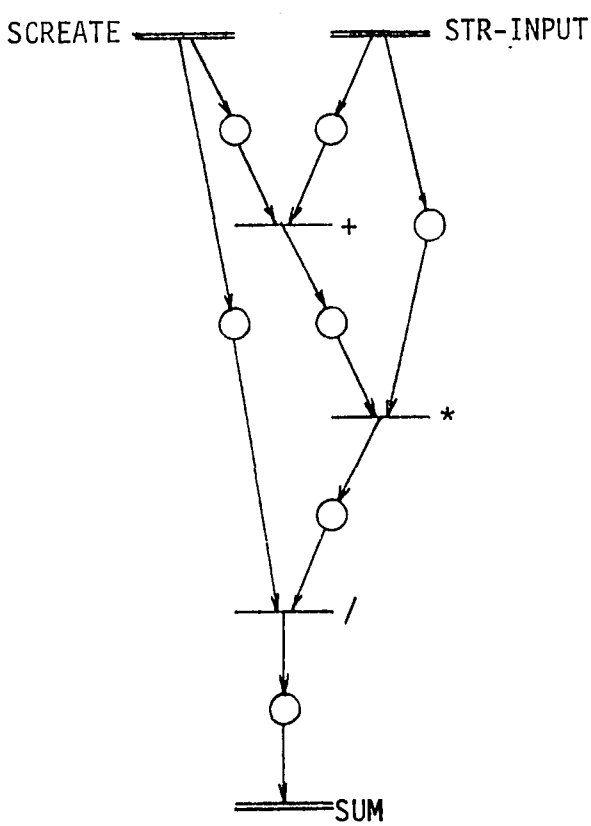


Figure 4.23. Petri net representation of streamed computation N of Figure 4.22

form and in graph form in Figure 4.23. This computation produces a value of $X = \Sigma_i\, i/((B_i + i) * B_i)$ where B_i is the ith token of the input stream B. The high level stream operations SCREATE and STR-INPUT serve as sources of the sequences of values and the SUM operation functions as a stream sink. These operations are pre-analyzed and appear as previously reduced nodes in Figure 4.23.

Attributes for nodes internal to streamed computations

The calculation of the nodal firing time of a streamed computation depends upon the attributes of previously reduced enclosed nodes N'. The reduction of a node internal to a streamed computation involves the computation of attributes characterizing its stand-alone timing behavior and its potential influence upon the periodic behavior of the streamed computation. These attributes are $t_1(N')$, $t_2(N')$, $n(N')$, $IP(N')$, entry(N'), and exit(N') and are described in subsequent paragraphs with references to the above example.

The value of t_1 for a node internal to a streamed computation represents the time required, once the node is initiated, to produce its first (and perhaps only) output set. For a stream source this value represents the time to produce its first stream token (of each output stream, should more than one be produced) given the availability of its scalar inputs. In the current example, assume

t_1(SCREATE) = 5 and t_1(STR-INPUT) = 3. The t_1-value for a stream sink denotes the time required to produce its scalar output value(s) given the first token of its input stream (and all scalar inputs). For stream sinks, the expression of this value usually involves the period p of its input stream; e.g., t_1(SUM) = 2 + m · max[p,3] where m is the length of the input stream. The constant portion of this expression, 2, relates the number of time steps required for SUM to produce its output value once it receives the eos token of its input stream (see Appendix A). The average time delay between the availability of the tokens of the stream input to the SUM node is captured in the p term. The value of 3 in t_1(SUM) points out that, due to the internal structure of the SUM template (see Appendix A), SUM may accept stream tokens at a maximum period of 3 time steps. For nodes that comprise a phase of the streamed computation by taking successive input stream values and producing successive output stream values (hereafter referred to as "phase" nodes), the t_1-value represents the time needed to produce their first output values given their first input values.

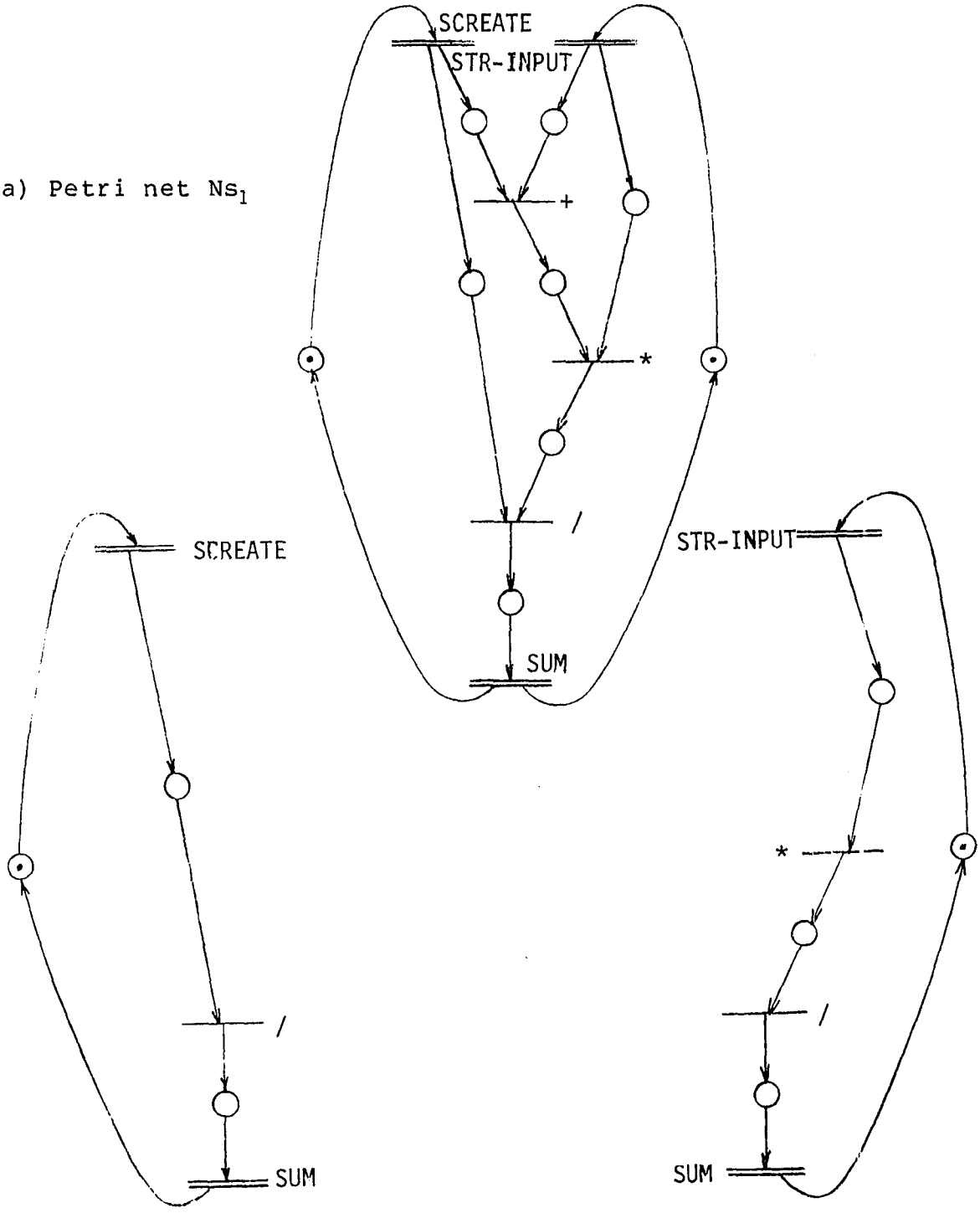
From the Petri net representation of the streamed computation N, the Petri net Ns_1 is constructed in the normal fashion by the addition of acknowledge places initialized with one token from each exit transition to each

entry transition. The period of this SMD Petri net corresponds to the nodal execution time of the streamed computation. For the example of Figures 4.22 and 4.23, the Petri net Ns_1 and its state machine decomposition appear in Figures 4.24. From this decomposition is computed

$$
\begin{aligned}
t_1(N) &= \tau(Ns_1) \\
&= \max[t_1(\text{SCREATE}) + t(/) + t_1(\text{SUM}), \\
&\qquad t_1(\text{SCREATE}) + t(+) + t(*) + t(/) + t_1(\text{SUM}), \\
&\qquad t_1(\text{STR-INPUT}) + t(+) + t(*) + t(/) + t_1(\text{SUM}), \\
&\qquad t_1(\text{STR-INPUT}) + t(*) + t(/) + t_1(\text{SUM})] \\
&= \max\{5 + 1 + (2 + m \cdot \max[p,3]), \\
&\qquad 5 + 1 + 1 + 1 + (2 + m \cdot \max[p,3]), \\
&\qquad 3 + 1 + 1 + 1 + (2 + m \cdot \max[p,3]), \\
&\qquad 3 + 1 + 1 + (2 + m \cdot \max[p,3]\} \\
&= 10 + m \cdot \max[p,3] \qquad (4.6)
\end{aligned}
$$

The period p of a stream input to a node within a streamed computation is determined in part by the internal behavior of nodes within the streamed computation and by a potentially complex interaction between these nodes. A reduced node N' internal to the streamed computation N may have a structure such that its independent period, IP(N'), determines the period of this stream. The computation of an IP-value assumes the total availability of the inputs to N' (including entire streams if appropriate) and the ability to release outputs as soon as they are produced. For source

(a) Petri net Ns_1



(b) State machine decomposition of Ns_1

Figure 4.24. Petri net Ns_1 and its SMD

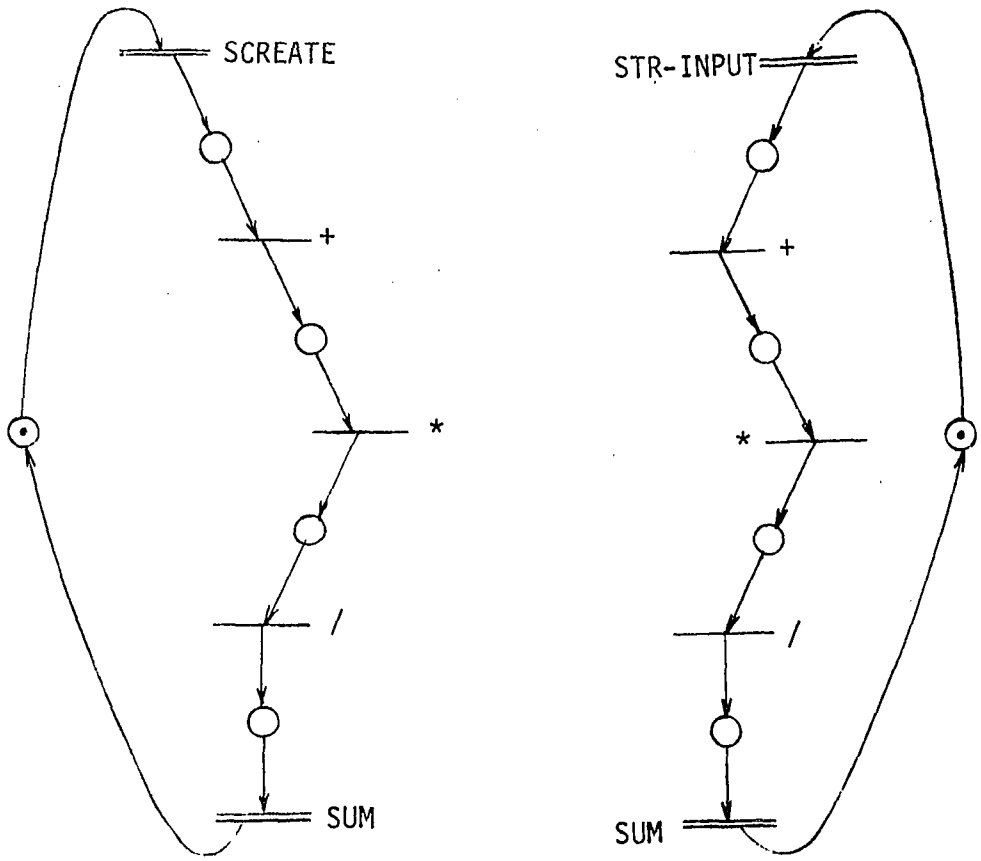


Figure 4.24. Continued

nodes N', IP(N') denotes the period between generation of successive output tokens. For stream sinks N', IP(N') is the period between successive absorptions of the tokens of the input stream. For stream sinks, this value is incorporated into the value of t_I(N') (IP(SUM) = 3 as pointed out previously). For phase nodes, IP(N') is the period between successive output tokens of the stream(s) given the availability of the input stream(s).

The interaction between nodes of a streamed computation

may be influenced by four values for each node N'. The first is the degree of pipelining, n(N'), possible within each node and is computed as the minimum number of internal stages that the node supplies to an enclosing stream computation. The second, $t_2(N')$, represents the maximum time for a single set of input values (one token from each input stream) to pass through the stages of N' once the periodic behavior of N' has been established. The remaining two attributes, entry(N') and exit(N'), are the maximum firing times of the entry and exit transitions, respectively, of N'. As will be noted later, all attributes are not necessary for all nodes.

The Petri net Np is constructed from N to model the complex interaction between the enclosed nodes of the streamed computation that may influence the period of the stream input to a given node. Np introduces acknowledge arcs to model the prevention of overwrite of successive values of the stream. This is accomplished by feedback signals in the underlying architecture. All "scalar" operations and all data arcs used for the transferral of scalar values in N are removed and each data arc of N' used for stream propagation is replaced by an acknowledge/data arc pair with the acknowledge place initialized with one token. For example, the Petri net Np of Figure 4.25 is constructed from the Petri net N of Figure 4.23 to compute the period of the stream input to the SUM node.

In the state machine decomposition of Np there may exist four types of state machines that involve a previously reduced node N':

I. state machines that have a path passing through N' in a "forward" direction (from an entry transition to an exit transition),

II. state machines that have a path passing through N' in a "backward" direction (from an exit to an entry transition),

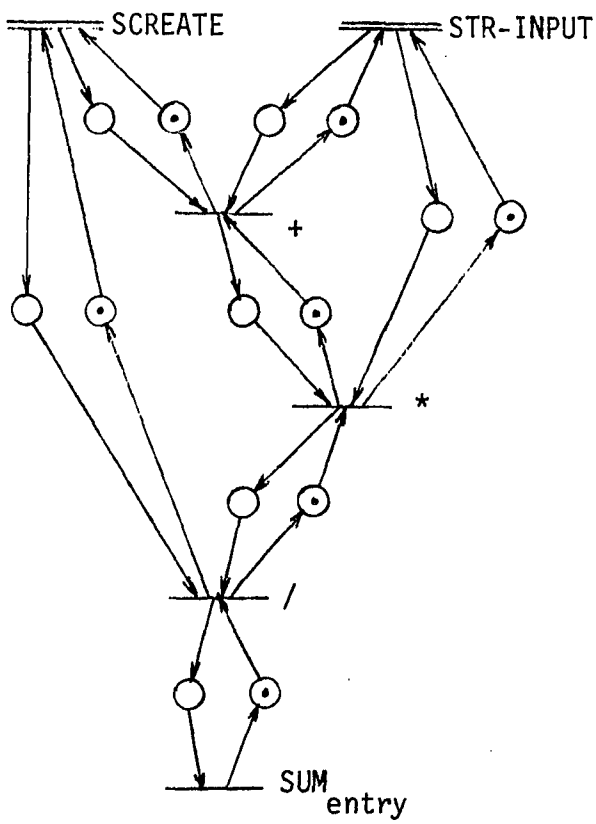


Figure 4.25. Petri net Np

III. state machines that involve only an entry transition of N', and

IV. state machines that involve only an exit transition of N'.

Figure 4.26 illustrates this for a hypothetical net Np. The computation of p is given as

$$p = \max[\tau(Np), IP(N_1), IP(N_2), \ldots, IP(N_X)] \qquad (4.7)$$

where Np contains x previously reduced nodes. $\tau(Np)$ is computed by Equations (3.1) and (3.2) where a previously reduced node N' makes the following contributions (in Equation (3.2)) to the period of state machines of the following type with respect to N':

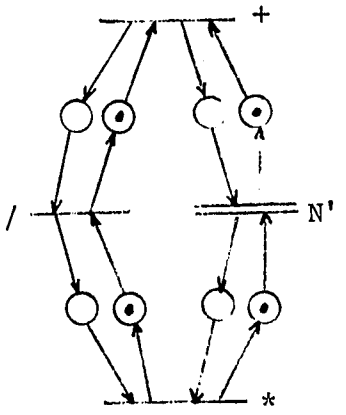
type I: N' contributes $t_2(N')$ to the numerator

type II: N' contributes $t_2(N')$ to the numerator and $n(N')$ to the denominator

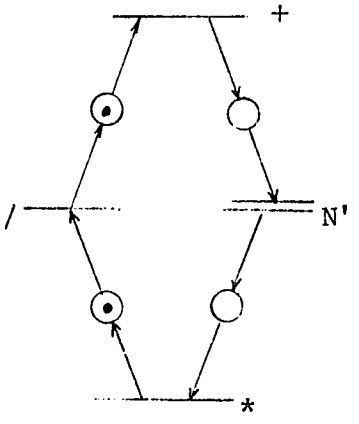
type III: N' contributes entry(N') to the numerator

type IV: N' contributes exit(N') to the numerator.

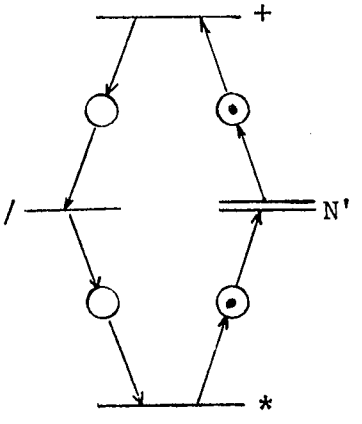
As an example, assume that the previously reduced node N' in Figure 4.26 has the following attributes: $t_2(N') = 8$, $n(N') = 2$, $IP(N') = 6$, entry(N') = 1, and exit(N') = 1. Furthermore, assume unit execution time for all base level operations. Np is decomposed into the state machines N_1 - N_6 of Figure 4.26(b) and



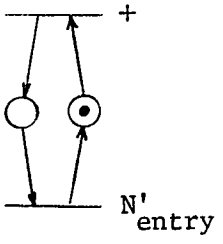
(a) Petri net Np with previously reduced node N'

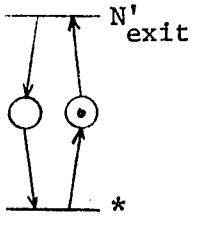


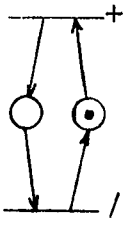
(i) type I

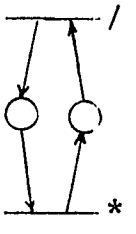


(ii) type II









(iii) type III (iv) type IV (v) remaining state machines

Figure 4.26. Petri net Np and its SMD

$$p = \max[\tau(Np), IP(N')]$$
$$= \max[\tau(N_1), \tau(N_2), \tau(N_3), \tau(N_4), \tau(N_5), \tau(N_6), IP(N')]$$
$$= \max[11/2, 11/4, 2/1, 2/1, 2/1, 2/1, 6] = 6$$

Likewise, the state machine decomposition of the Petri net Np of Figure 4.25 appears in Figure 4.27 and the period of the stream input to the SUM operation is computed as

$$p = \max[\tau(Np), IP(SCREATE), IP(STR\text{-}INPUT)].$$

Note that IP(SUM) is not included as this value is incorporated into t_1(SUM). Assuming IP(STR-INPUT) = 5, IP(SCREATE) = 4, exit(STR-INPUT) = 1, exit(SCREATE) = 1, and entry(SUM) = 1, the calculation proceeds as

$$p = \max[\tau(N_1), \tau(N_2), \ldots, \tau(N_{13}), 4, 5]$$
$$= \max[4/1, 4/3, 3/1, 3/2, 5/2, 5/3, 2/1, 2/1, 2/1,$$
$$2/1, 2/1, 2/1, 2/1, 4, 5]$$
$$= 5$$

This value is used in conjunction with Equation (4.6) to yield

$$t_1(N) = 10 + m \cdot \max[5,3]$$
$$= 10 + 5(m)$$

where N is the streamed computation of Figures 4.22 and 4.23.

Several points should be made about the nature of Np-type Petri nets: first, not all acknowledge arcs are necessary in Np to guarantee the safety of data tokens [Brock and Montz 1979]. Second, the balancing of the

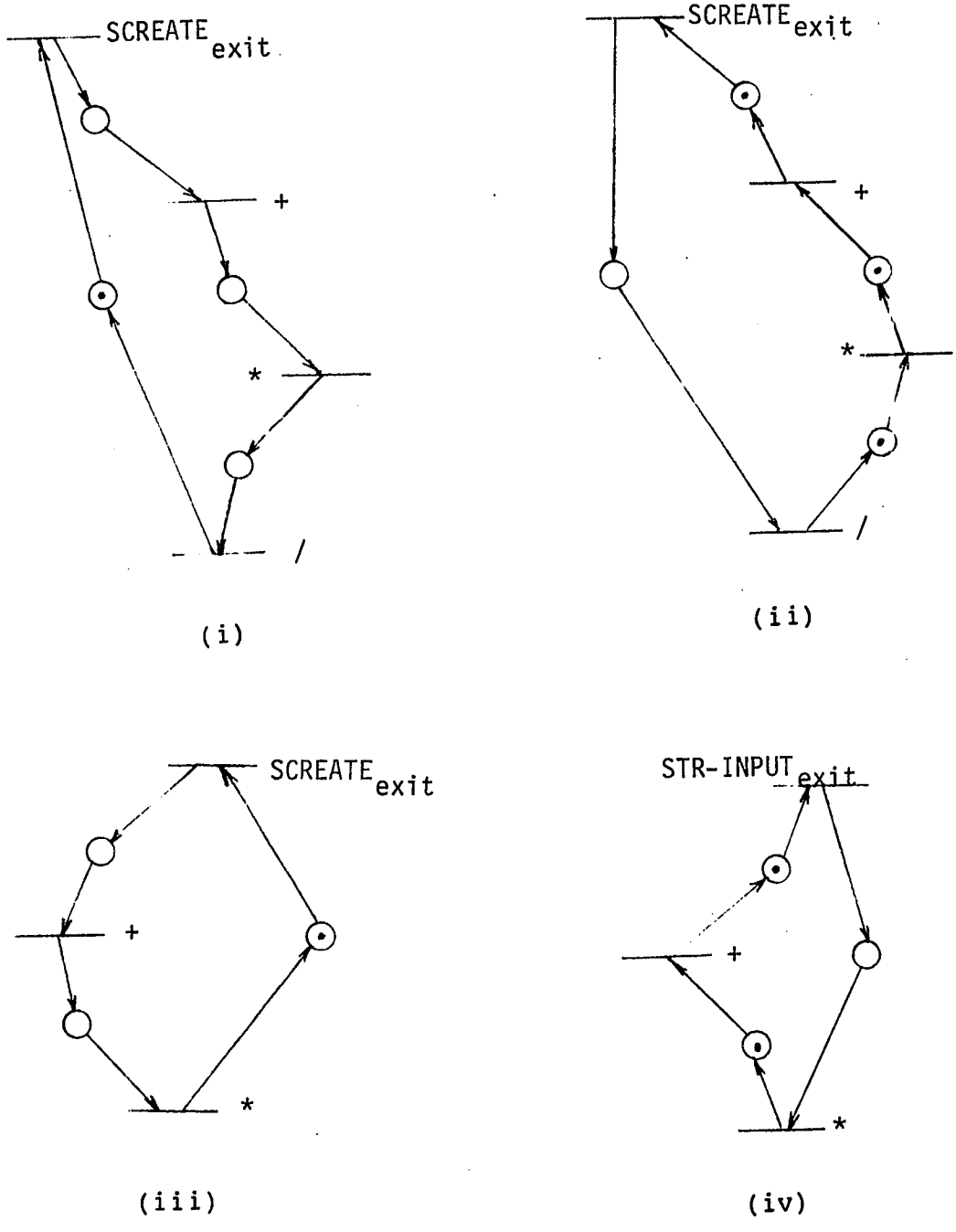


Figure 4.27. State machine decomposition of the Petri net Np of Figure 4.25

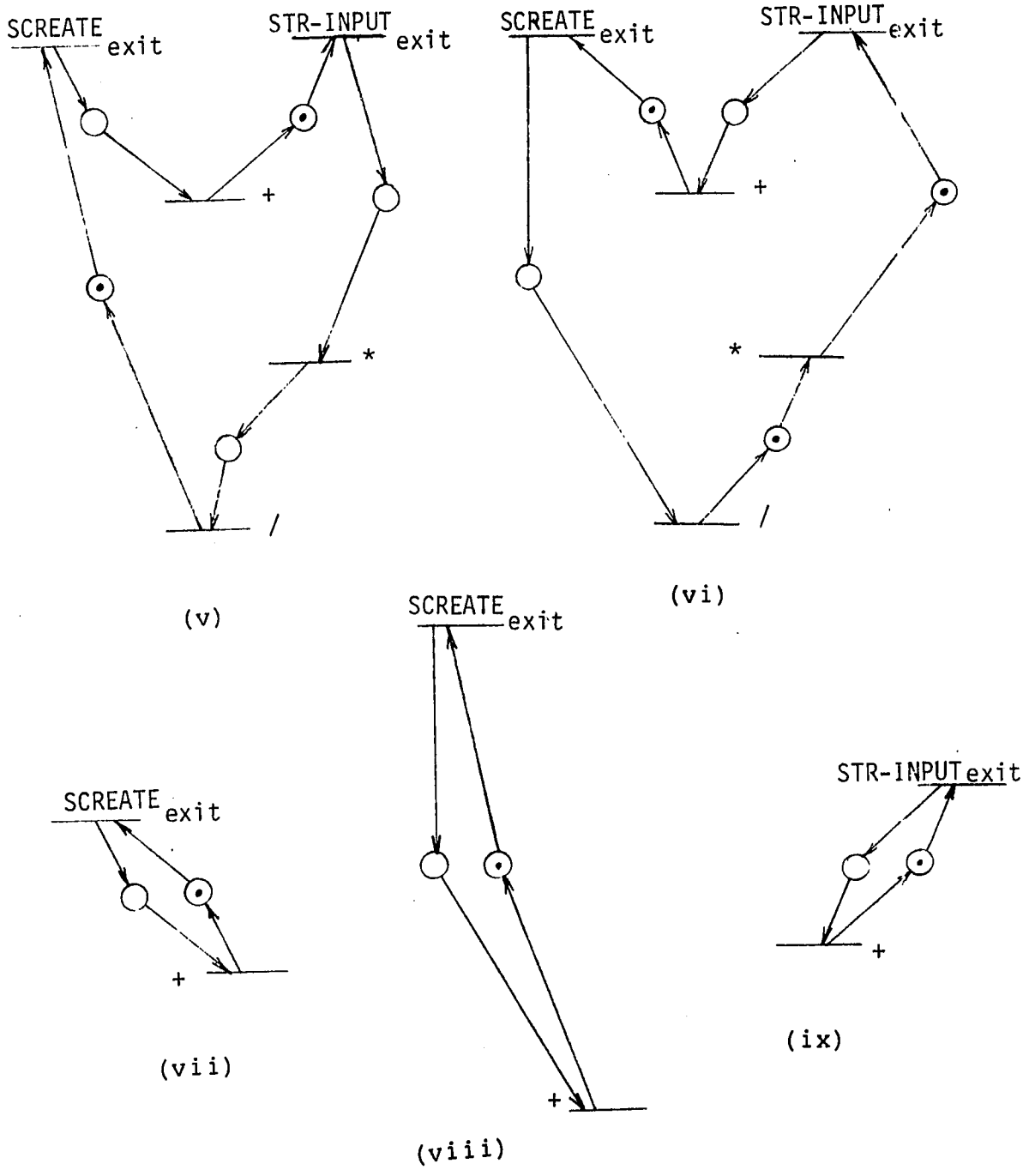


Figure 4.27. Continued

program graph through the insertion of "identity" operations may optimize the bahavior of the program [Brock and Montz 1979]. However, neither of these aspects affect the general method of the analysis of programs and will not be given further consideration. Additionally, it is possible to syntactically include in a streamed computation scalar computations. (This somewhat defeats the purpose of the declaration at the high level of the streamed computation N.) However, since "scalar" computations and scalar arcs are removed in the construction of Np, these computations have no bearing on the calculation of the period of the streams.

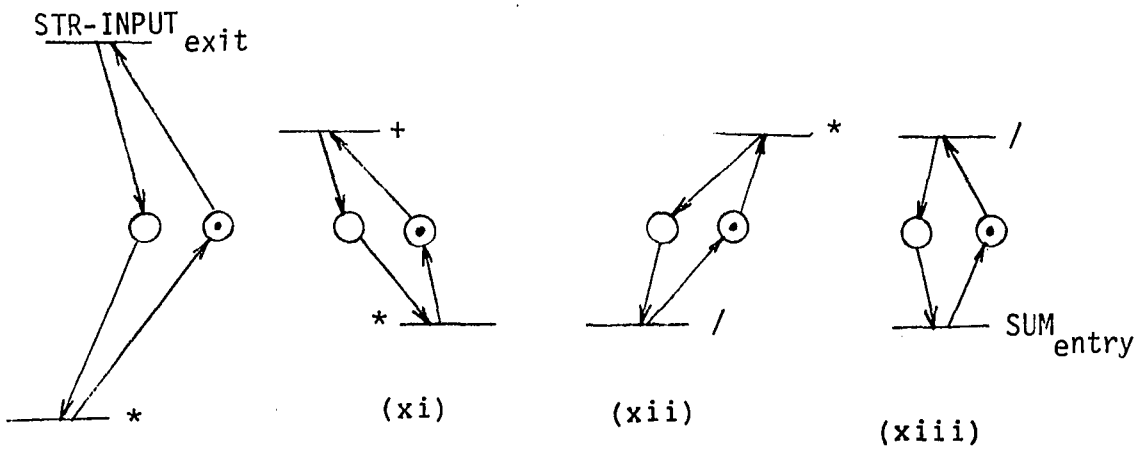


Figure 4.27. Continued

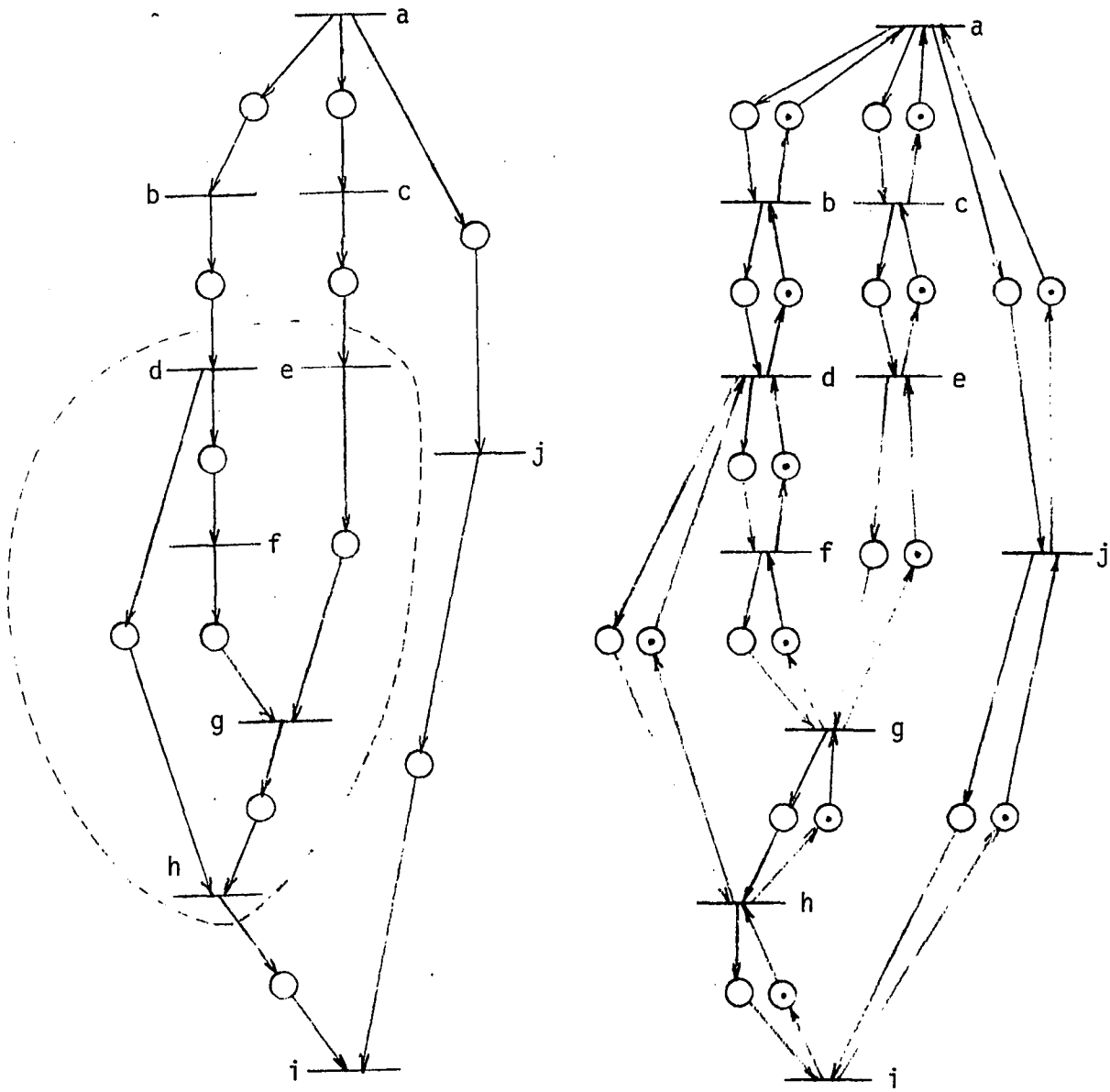
The role of the independent period IP(N') of a reduced node N' in an Np-type Petri net is further illustrated by the Petri net N of Figure 4.28(a). The Np-type Petri net constructed directly from N is shown in Figure 4.28(b). Should, however, the portion of N within the dotted line be recognized as a node N' and reduced as such, N appears as the Petri net M of Figure 4.28(c) and the corresponding Np-type Petri net appears as Mp of Figure 4.28(d). The period of Mp should be an approximation of the period of Np. In the state machine decomposition of Np, there potentially exist nine types of state machines with respect to the transitions of the non-reduced node N' (transitions d through h):

a) state machines passing through N' in a forward direction (following forward arcs from transitions d or e to transition h),

b) state machines passing through N' in a backward direction (following backward arcs from transition h to transitions d or e),

c) state machines involving only an entry transition (transitions d or e),

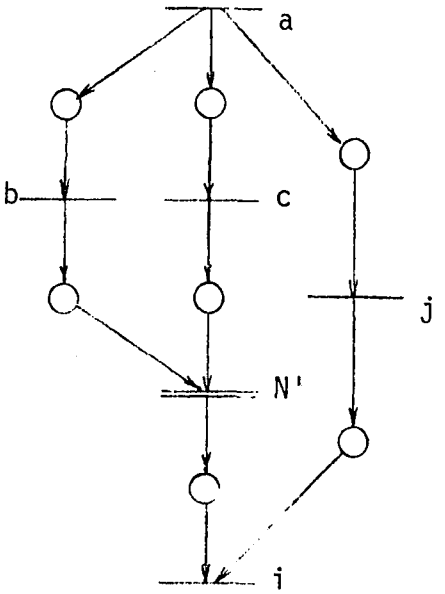
d) state machines involving only an exit transition (transition h),



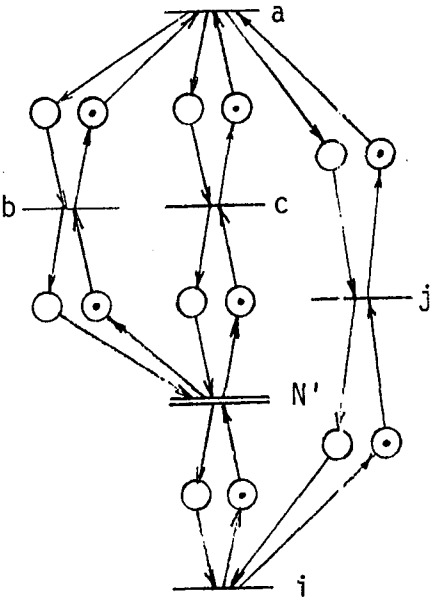
(a) Petri net N

(b) Petri net Np

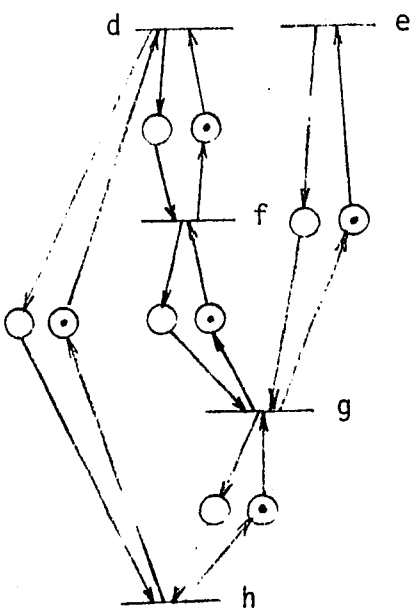
Figure 4.28. Nodal reductions in Np-type Petri nets



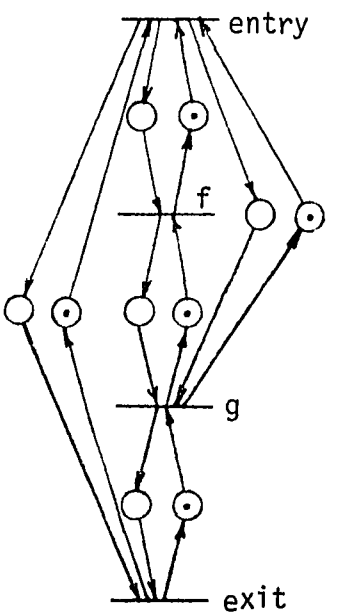
(c) Petri net M



(d) Petri net Mp

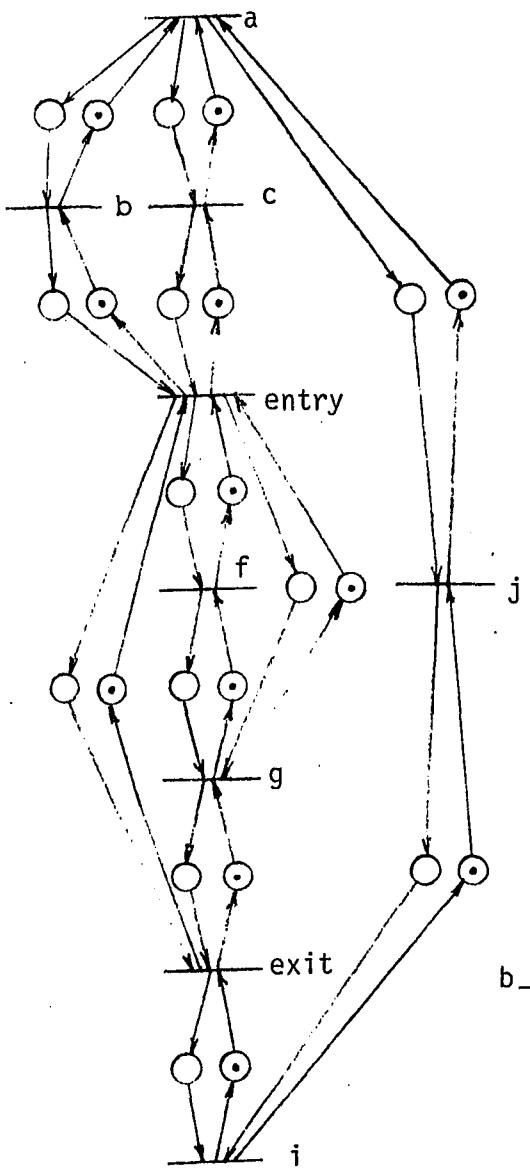


(e) initial representation of Petri net Np'

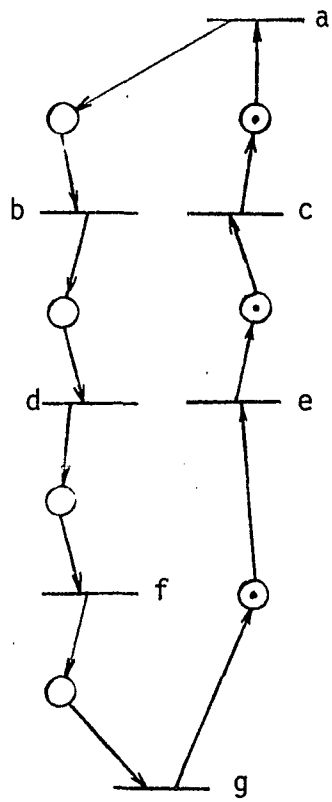


(f) modified representation of Petri net Np'

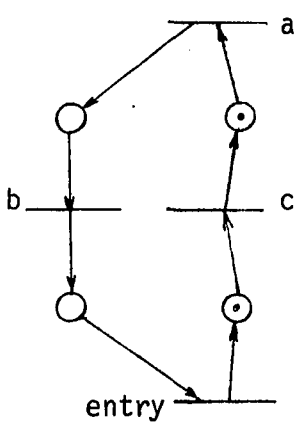
Figure 4.28. Continued



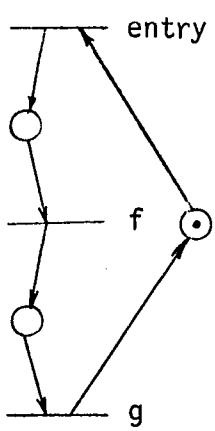
(g) expanded representation of Mp



(h) state machine N_1 of Np



(i) state machine N_2 of Mp



(j) state machine N_3 of Np'

Figure 4.28.
Continued

e) state machines entering N' through an entry transition and leaving N' through another entry transition,

f) state machines entering N' through an exit transition and leaving N' through another exit transition,

g) state machines containing only transitions of N',

h) state machines not containing any transitions of N', and

i) state machines involving multiple occurrences of types (a) through (f).

As mentioned earlier there are potentially four types of state machines (types I through IV) that may involve the reduced node N' of Mp. These four types of state machines provide an approximation of the state machines in Np of types (a) through (d) respectively. State machines of type (h) are also found unmodified by the reduction of N' in Mp. Generally speaking, IP(N') incorporates the state machines of type (g). An initial representation of the Petri net Np' is found in Figure 4.28(e). This Petri net contains exactly those state machines also found in Np of type (g). Two approaches can be taken to handle state machines of types (e), (f), and (i). The first is to ignore their influence. Since this approach may ignore the critical cycle, a second approach is taken: in the construction of the Petri net Np'

all entry transitions are coalesced into a single transition with all the input and output arcs of the entry transitions. The same occurs for all exit transitions (this is unnecessary for this example). The modified Petri net Np' appears in Figure 4.28(f). The period of this Petri net is used in the calculation of a value for IP(N'). This value is used in conjunction with τ(Mp) in the approximation of the period of the stream. This is equivalent to the use of the expanded Petri net Mp of Figure 4.28(g) as an approximation of the Petri net Np. However, the work required to decompose the Petri nets of Figures 4.28(d) and 4.28(f) separately is no more (and may be substantially less) than the work required to decompose the Petri net of Figure 4.28(g). In general, nodal reductions may provide a substantial reduction in work, especially when there exists a high level of nesting of nodes.

By coalescing entry and exit transitions in Np' there exist state machines to provide approximations for the state machines of Np of types (e), (f), and (i). For example, consider the state machine of type (e) in Np found in Figure 4.28(h). The period of this state machine is approximated by the period of the state machines found in Figure 4.28(i) and 4.28(j) of Petri nets Mp and Np', respectively. Since the Petri net of Figure 4.28(g) is a constrained representation of the Petri net of Figure 4.28(b) (by Constraint 2),

it is true that its period is not less than that of Np. This is due to the fact that

$\tau(N_1) < \max[\tau(N_2), \tau(N_3)]$, because

$7/3 < \max[4/2, 3/1]$.

A similar relationship holds for state machines of types (f) and (i) and the state machines approximating them.

High level stream templates The attributes of the high level stream templates described in Chapter II appear in Figure 4.29. All entry and exit attributes are assumed to be 1 since all base level operations are assumed to require unit execution time. The actual templates used in the data flow compiler used in this study for this work appear in Appendix A. These nodes are assumed to be pre-analyzed and serve as primitive nodes. These operations when modeled as Petri nets are often non-SMD. However, the underlying semantics (denoted by "labeled" tokens in the templates) of the operations used in this study impose a totally predictable behavior allowing attributes to be pre-specified.

The REST operation is a phase node and removes the first token from its input stream. For this reason, its t_1-value is expressed in terms of the delay $d_{1,2}$ between its first and second tokens:

$$t_1(\text{REST}) = 2 + \max[3, d_{1,2}].$$

However, a simplifying assumption reduces this to

$$t_1(\text{REST}) = 2 + \max[3, p]$$

where p is the period of its input stream.

The CONS operation is used to prefix a stream with a scalar value of the appropriate type and is especially useful in conjunction with streamed recursive procedures. The period p of its output stream is expressed in terms of

	t_1	t_2	IP	n
BUF	3 + m·max(4,p)	--	4	--
CONS	1	1	2	Ø
EMPTY	3	--	3	--
FIRST	2	--	3	--
REPL	4	4	3	2
REST	2 + max(3,p)	2	3	1
SCREATE	5	--	4	--
SELECT	6	6	2	5
STR-INPUT	3	--	5	--
STR-SWITCH	2	1	3	Ø
STR-MERGE	1	1	3	Ø
SUBSTM	7 + k·max(3,p)*	5	3	4
SUM/PROD	2 + m·max(3,p)	--	3	--
UNBUF	5	--	4	--

* - k denotes the number of tokens removed from the front of the stream

Figure 4.29. Attributes of high level stream operations

the period p' of its input stream and the delay $d_{1,2}$ between its first output token (its scalar input) and its second output token (its first input stream token):

$$p = \max[(d_{1,2} + m \cdot p')/(m + 1),\ IP(CONS)] \qquad (4.8)$$

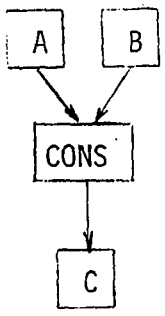
In most instances, the assumption that $d_{1,2} = p$ will be made for simplifying purposes. However, consider the node N found in Figure 4.30(a) where A is the abstract operation(s) that computes the scalar input to the CONS operation, B is the abstract operation(s) associated with the input stream, and C is the abstract operation(s) involving the output stream of the CONS operation. Treating A, B, and C as nodes, the Petri net Ns_1 of Figure 4.30(b) is constructed to compute the value:

$$t_1(N) = \tau(Ns_1) = t_1(A) + t_1(CONS) + t_1(C) \qquad (4.9)$$

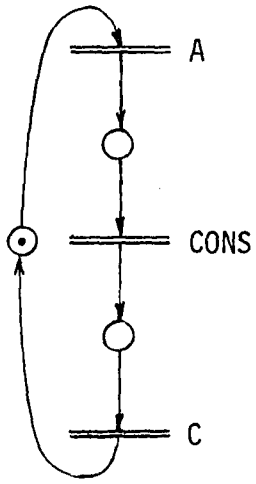
Furthermore, the delay between the first and second tokens output from N may be computed as:

$$d_{1,2} = \max\{\max[\tau(Np'),\ IP(CONS),\ IP(C)],\ \tau(Ns') - t_1(N)\} \qquad (4.10)$$

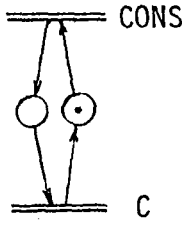
The first argument of the outermost maximum function states that the delay between the first and second tokens is subject to the constraining influences of the state machines through which both tokens must pass (see Figure 4.30(c)). For the computation of $\tau(Ns')$, the Petri net Ns' of Figure 4.30(d) is constructed from N in the usual fashion but



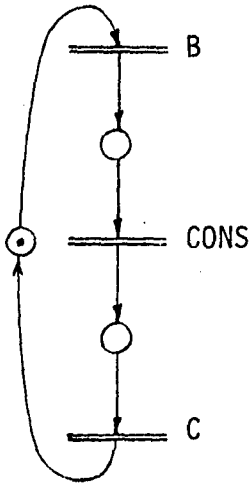
(a) Node N with CONS operation



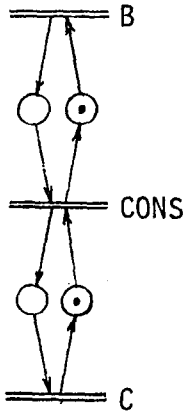
(b) Petri net Ns_1 for $t_1(N)$



(c) Petri net Np'



(d) Petri net Ns'



(e) Petri net Np

Figure 4.30. Petri nets used in analysis of streamed node with CONS operation

ignores the scalar computation (A). Since the first token of the output stream (the scalar input to the CONS operation) has already passed through the CONS operation and the node C, these nodes have a nodal firing time of $t_2(CONS)$ and $t_2(C)$, respectively. The value for $\tau(Ns')$ is therefore computed as

$$\tau(Ns') = t_1(B) + t_2(CONS) + t_2(C) \qquad (4.11)$$

This value is the time required before the second output token would be available, assuming the first token did not hold it up. Any delay due to the unavailability of this token is captured in the term $\tau(Ns') - t_1(N)$ in Equation (4.1Ø). Noting that $t_1(CONS) = t_2(CONS)$ and combining Equations (4.1Ø) and (4.11):

$$\begin{aligned} d_{1,2} &= \max[\tau(Np'), IP(CONS), IP(C), \tau(Ns') - t_1(N)]] \\ &= \max\{exit(CONS) + entry(C), IP(CONS), IP(C), \\ &\qquad [t_1(B) + t_2(CONS) + t_2(C)] - [t_1(A) + \\ &\qquad t_1(CONS) + t_1(C)]\} \\ &= \max[1 + 1, 2, IP(C), \\ &\qquad t_1(B) + t_2(C) - t_1(A) - t_1(C)] \\ &= \max[2, IP(C), \\ &\qquad t_1(B) - t_1(A) + t_2(C) - t_1(C)] \end{aligned} \qquad (4.12)$$

The period p' of the remainder of the output stream of N is calculated in the usual fashion in conjunction with the Petri net Np of Figure 4.3Ø(e):

$$p' = \max[\tau(Np), IP(B), IP(CONS), IP(C)]$$
$$= \max[exit(B) + entry(CONS), exit(CONS) + entry(C), IP(B), 2, IP(C)]$$
$$= \max[1 + 1, 1 + 1, IP(B), 2, IP(C)]$$
$$= \max[2, IP(B), IP(C)] \qquad (4.13)$$

The period p of the output stream of N, therefore, becomes (using Equations (4.12) and (4.13)):

$$p = (d_{1,2} + m \cdot p') / (m + 1)$$
$$= \{\max[2, IP(C), t_1(B) - t_1(A) + t_2(C) - t_1(C)] + m \cdot \max[2, IP(B), IP(C)]\} / (m + 1) \qquad (4.14)$$

The equation and the construction of appropriate Petri nets is similar for nodes involving multiple CONS operations.

The FIRST operation in Figure 4.29 is a stream sink that consumes its entire input stream. However, its result (a scalar value) is available two time steps after the arrival of the first token of its input stream, hence,

$$t_1(FIRST) = 2.$$

If, however, the streamed computation of which this operation is a part is directly embedded in another construct that will reuse the streamed computation's token space (e.g., another streamed computation or an iterative while-do), then the re-invocation of the streamed computation requires that the previous stream be consumed before a re-invocation may take place (except for a negligible overlap). For this reason, stream sinks whose results are

available before the entire stream has been processed have alternate expressions of their t_1-value. For the FIRST operation, this is

$$t_1(\text{FIRST}) = 2 + m \cdot \max[p,3]$$

where p is the period of the stream input to the FIRST node. This alternate expression is used in those cases where the token space is reused which is determined at the time the enclosing node is reduced.

Attributes for a streamed computation node In general, any type of node N' may appear within a streamed computation N. For this reason, it is necessary to ascribe attributes for those nodes. The scalar nodes previously described (blocks, conditionals, while-do, scalar procedures, and streamed computations) all take as input and produce as output scalar values. Should these scalar values be individual tokens of a stream (syntactically expressed through the association entry), then for the majority of the computation only one token from the enclosing stream computation has been absorbed. As a consequence, $n(N') = 1$ provides a very tight lower bound and

$$t_2(N') = IP(N') = t_1(N').$$

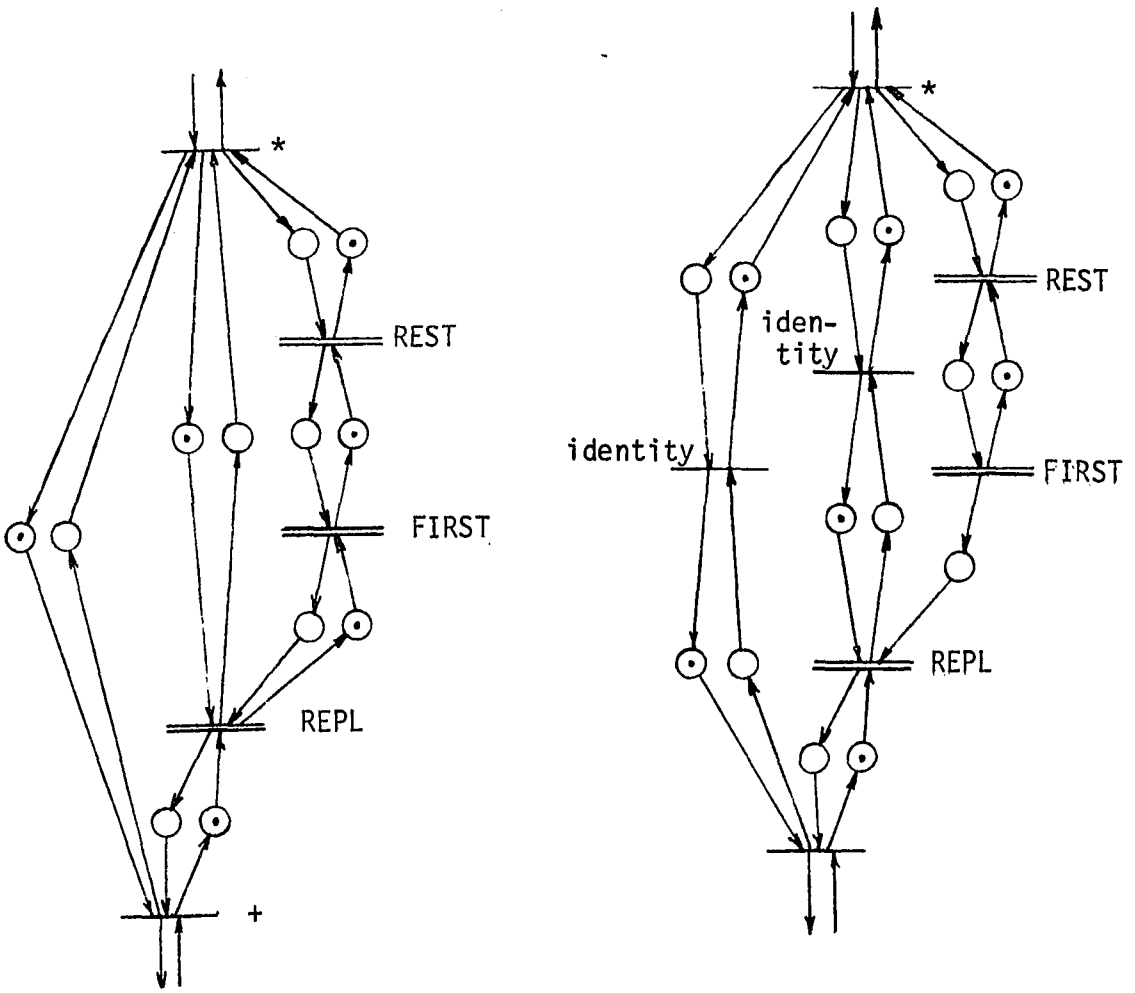
For any node, the entry attribute is recursively defined as

$$\text{entry}(N') = \max_i [\text{entry}(E_i)] \qquad (4.15)$$

where E_i is an entry node of N'. Should E_i be a base level operation, $\text{entry}(E_i)$ is its transition firing time.

exit(N') is described in a similar fashion. Since the assumption that all base level operations have unit execution time is made, entry(N') = exit(N') = 1 for all nodes N'.

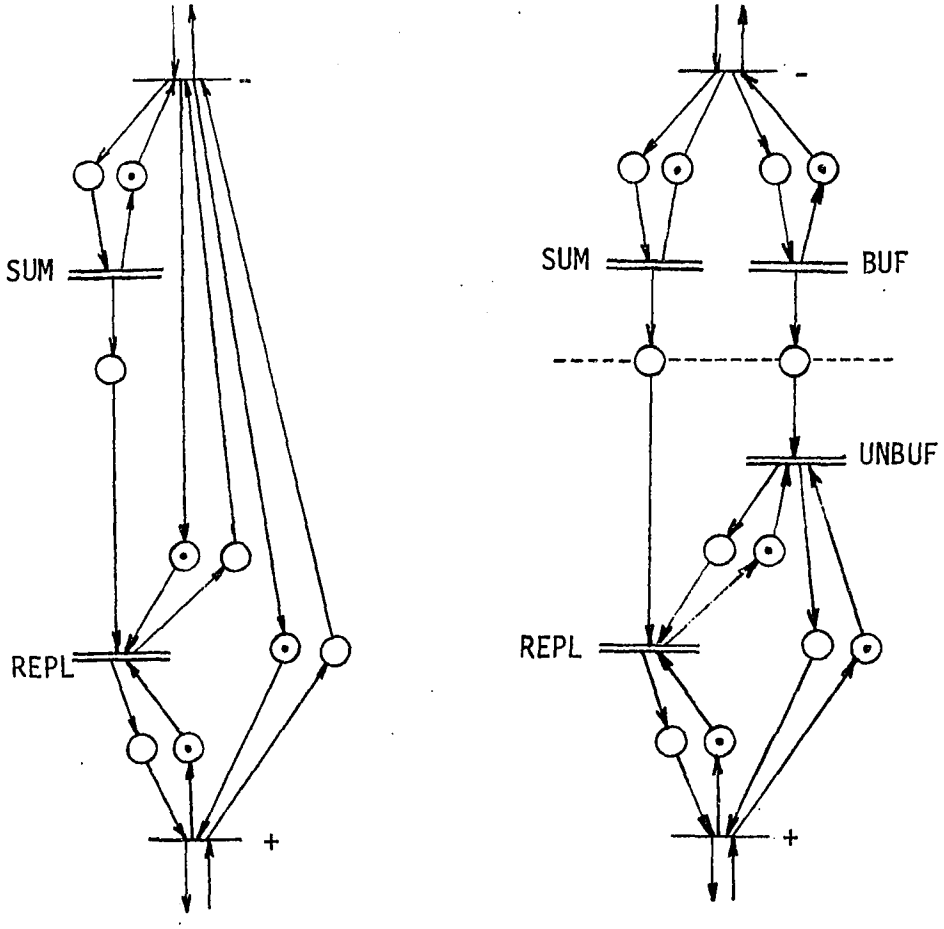
Program graphs and deadlock Since the token space for the elements of a stream lies partially on the program graph, care must be taken to prevent situations where deadlock might occur. For example, Figure 4.31(a) shows a portion of a streamed computation that is deadlocked. In this example the first token of the stream produced by the * operation has been passed to the +, REPL, and REST operations. The REST operation has consumed its first token (without producing a result) and as a result, the Petri net of Figure 4.31(a) is deadlocked. This situation is remedied by the insertion of identity operations as shown in Figure 4.31(b). Other situations require the buffering (and unbuffering) of the entire stream into (from) array storage (see Figure 4.32). As the result of the buffering and unbuffering as found in Figure 4.32(b), two distinct streamed computations are created; the first (above the dotted line) provides scalar inputs to the second (below the dotted line). It is also possible to write at the high level situations where nodes are not strictly hierarchal. For example, Figure 4.33 shows the high level code where two streamed computations have been placed in a single streamed



(a) deadlocked graph (b) deadlock-free graph

Figure 4.31. Deadlock avoidance by "identity" insertion

construct. To make matters worse, portions of both are found within the procedure body. In this situation, it is necessary to split the procedure code so that the two distinct streamed computations may be analyzed separately. The detection of the need for buffering and unbuffering and



(a) deadlocked Petri net (b) deadlock-free Petri net

Figure 4.32. Deadlock avoidance by buffering

splitting of nodes is certainly not trivial. However, these concerns lie outside the scope of ths research. Consequently, the following assumptions are made:

1. it is possible to detect single and separate streamed computations,
2. buffering (and unbuffering) of streams into (from) array storage is appropriately placed in the code to be analyzed, and
3. nodes, as found in the code to be analyzed, are strictly hierarchal.

These restrictions are assumed to be met by the programmer, compiler, and/or pre-processor.

```
STREAMED
   INTEGER STREAM A,B,C;
   INTEGER Y
   PROCEDURE P (INTEGER STREAM A)
         RETURNS (INTEGER STREAM B)
      INTEGER X;
      X = SUM(A);
      B = SCREATE(1, X, 1)
      END;
   STR-INPUT A FROM FILE;
   B = P(A);
   Y = SUM(B)
   END (Y)
```

Figure 4.33. Two streamed computations in single construct

Blocks (streamed)

A streamed block N supplies a certain number of stages to the pipeline of the enclosing streamed computation. The nodal firing times are calculated in the usual fashion by the construction and analysis of Ns-type Petri nets:

$t_1(N) = \tau(Ns_1)$ and $t_2(N) = \tau(Ns_2)$

where Ns_1 and Ns_2 are obtained from N by the addition of acknowledge places initialized with one token each from all exit to all entry nodes. Ns_2 additionally has all source nodes (and nodes strictly data dependent upon source nodes) and all sink nodes (and nodes producing values solely for sinks nodes) and their connecting arcs removed. A previously reduced node N' internal to Ns_1 or Ns_2 is treated as a transition with a nodal firing $t_1(N')$ or $t_2(N')$, respectively.

The independent period, IP(N), of the block is computed as

$$IP(N) = \max[\tau(Np), IP(N_1), IP(N_2), \ldots, IP(N_x)] \quad (4.16)$$

where N contains x previously reduced nodes (compare with Equation (4.7)). The Petri net Np is constructed in the usual fashion by eliminating all scalar operations and associated arcs and replacing all stream arcs by acknowledge/data arc pairs with the acknowledge places initialized with one token each. As illustrated earlier, it is necessary to enforce Constraint 2 by coalescing all entry

and all exit transitions in Np whose inputs and outputs, respectively, are streams. Note that this does not include the nodes of N that function as stream sources (or stream sinks) since their inputs (or outputs, respectively) are scalar values. When two previously reduced nodes, A and B, are coalesced into a node C, C has the maximum attributes of A or B; e.g.,

$$IP(C) = \max[IP(A), IP(B)].$$

The period of the stream input to a node N' internal to block N may be affected by the state machines of Np and the independent period of any other previously reduced nodes of N. The value of p in the attribute(s) of N' is, therefore, replaced by max[p', IP(N)] where p' is the period of the stream input to N.

n(N) is the number of stages that N contributes to the enclosing streamed computation and is computed as

$$n(Np) = \min_i[s(path_i)] \qquad (4.17)$$

where $path_i$ is a path from the exit transition to the entry transition of Np and

$$s(path_i) = \Sigma_j M(p_j) + \Sigma_k n(N_k) \qquad (4.18)$$

where p_j is a place in $path_i$ and N_k is a previously reduced node within N along $path_i$. This value is not relevant if N provides all the stream sources or all the stream sinks in the enclosing stream computation.

An example of a streamed block is found in Figure

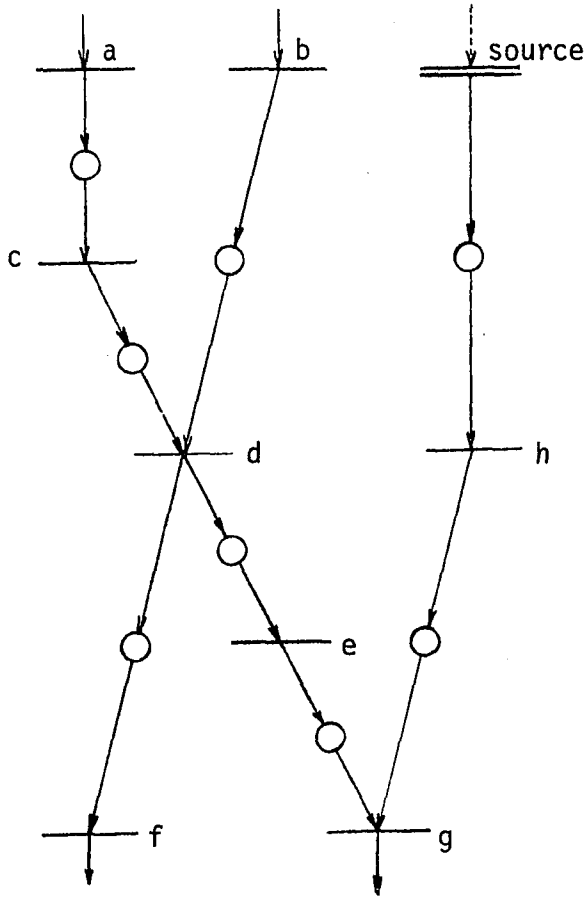
4.34(a). The attributes for this block are computed using the Petri nets Ns_1, Ns_2, and Np of Figures 4.34(b) through 4.34(d) as

$$t_1(N) = \max[5,\ 2 + t_1(\text{source})]$$

$$t_2(N) = 5$$

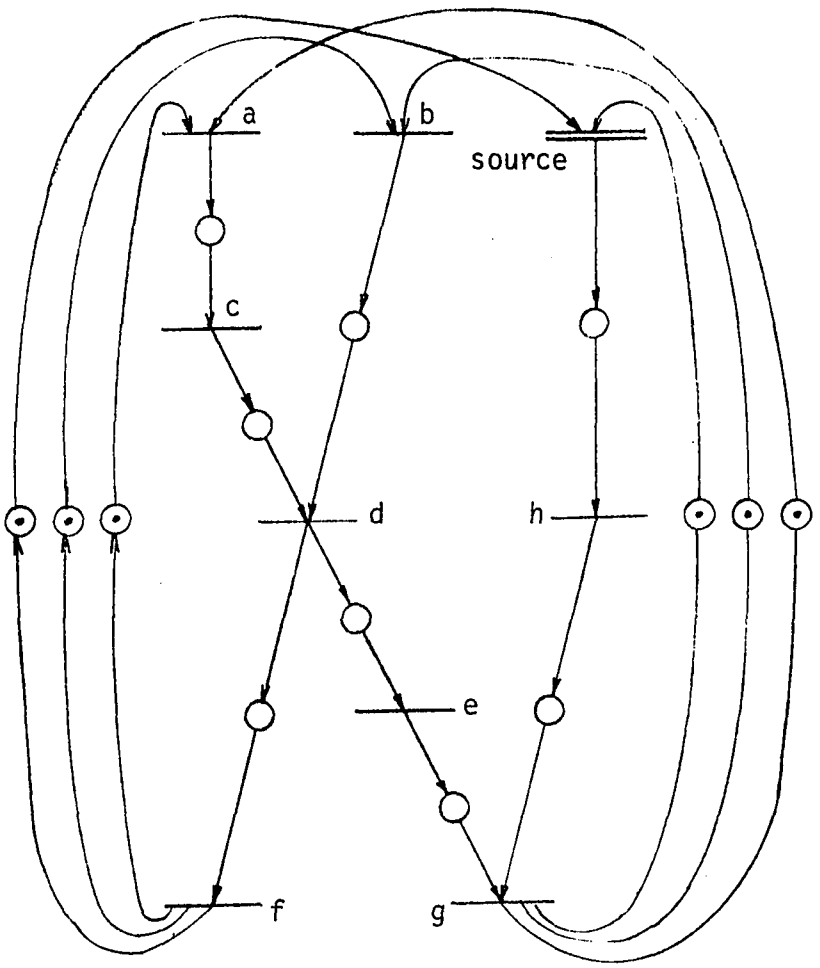
$$IP(N) = \max[\tau(Np),\ IP(\text{source})] = \max[3,\ IP(\text{source})]$$

$$n(N) = \min[4,\ 3,\ 3,\ 2] = 2$$



(a) Block N

Figure 4.34. Example block N



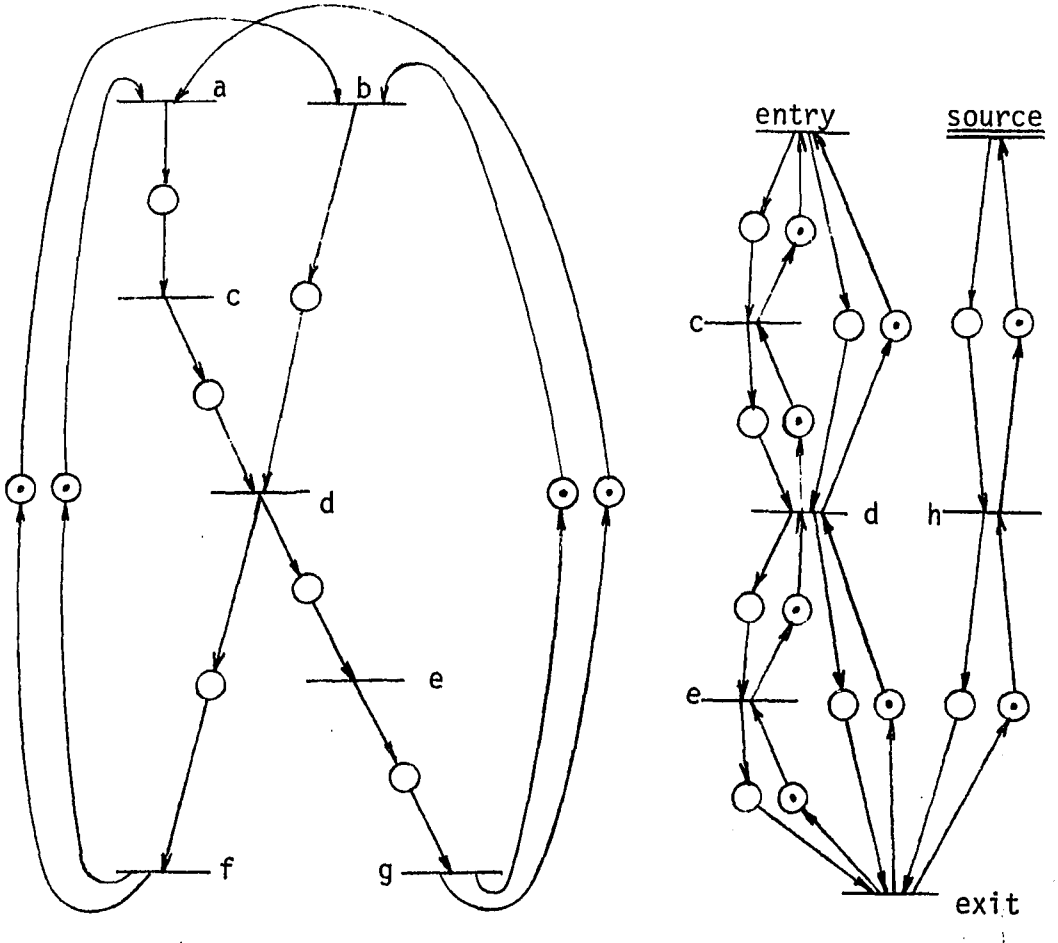
(b) Petri net Ns_1

Figure 4.34. Continued

Conditionals (streamed)

When streams are involved, two types of conditionals are possible. The first type arises when the boolean condition is evaluated once for every token of the input stream(s). This case presents the problem of determining

the optimal degree of balancing [Brock and Montz 1979]. An alternate approach is to implement this type of conditional with the high level stream operation SELECT described in a previous section. One possible template for this operation that is totally balanced appears in Appendix A.



(c) Petri net Ns_2 (d) Petri net Np

Figure 4.34. Continued

A decision is made to use the conditional for the second type where the condition is evaluated once and the SELECT for the other. Under this interpretation, the streams are routed entirely through the then body or the else body via STR-SWITCH and STR-MERGE operations. These appear as pre-analyzed nodes in Appendix A. Referring to the reduced conditional Petri net representations discussed in the section on scalar conditionals (see Figure 4.6), note that a streamed conditional is represented as found in

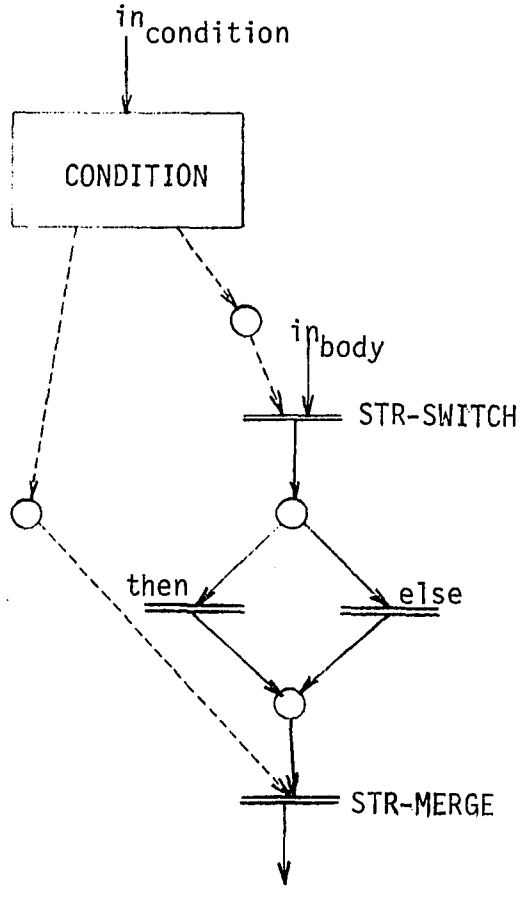


Figure 4.35. Petri net representation of streamed conditional

Figure 4.35. Scalar arcs are dotted; stream arcs are solid. The condition may be a complete scalar computation or it may "sink" an input stream to produce a scalar boolean gate value (e.g., by the statement: IF SUM(A) > Ø). There exist several strategies in the calculation of attributes of the streamed conditional. These are presented and discussed separately.

Strategy 1 Separate attributes are maintained for both possibilities of the conditional: if_t and if_f. For example, the attributes of if_t are computed as:

$$t_1(if_t) = \tau({}^tNs_1)$$

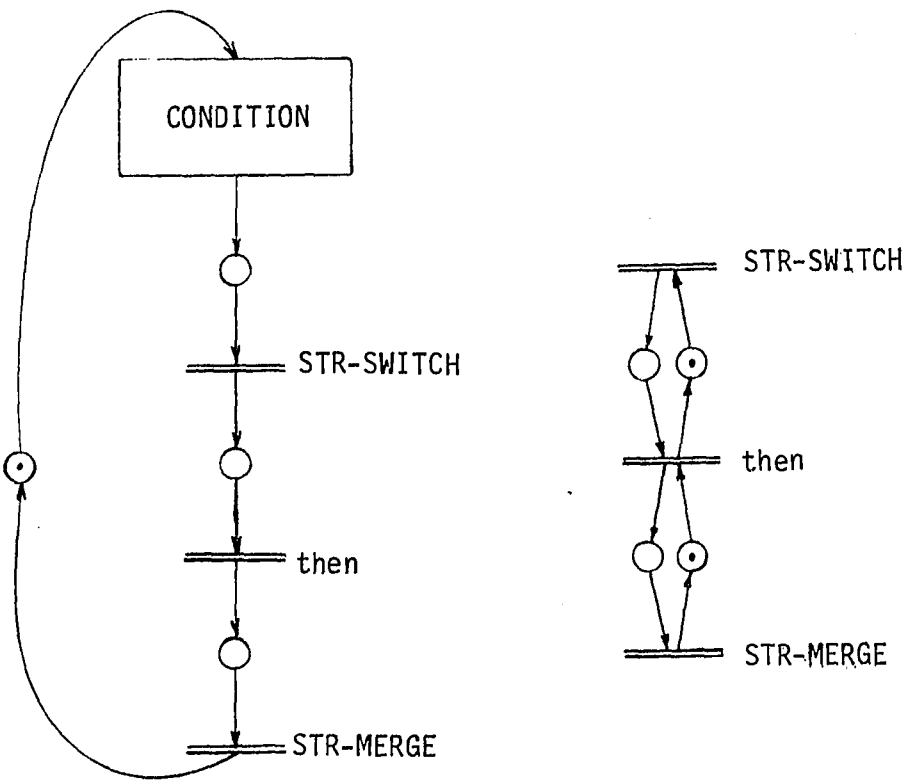
$$t_2(if_t) = \tau({}^tNs_2)$$

$$IP(if_t) = \max[IP(CONDITION), \tau({}^tNp), IP(STR\text{-}SWITCH), IP(then), IP(STR\text{-}MERGE)]$$

$$n(if_t) = 2 + n(then)$$

The ${}^tNs$-type Petri nets appear in Figure 4.36(a) and the Petri net Np appears in Figure 4.36(b). The condition is treated as a block and a value is computed for IP(CONDITION) in $IP(if_t)$. Should the evaluation of the condition only involve scalars this term is ignored. Since the condition necessarily sinks any input stream, and n(STR-SWITCH) = n(STR-MERGE) = Ø, $n(if_t)$ is computed as specified (when the condition provides at least two stages to the enclosing "pipelined computation). The attributes of if_f would be computed similarly. The main disadvantage with the approach

of strategy one is that the number of possible paths drastically increases with the number of conditionals in a streamed computation (2^x different paths for x non-nested conditionals). Furthermore, any node containing conditionals will also have multiple sets of attributes, one set assuming the if_t portion of the node and one set assuming the if_f portion of the node. Since the resulting analysis becomes arbitrarily complex (contrary to the analysis



(a) $^{t}N_s$-type Petri nets (b) Petri net $^{t}N_p$

Figure 4.36. Petri nets for if_t

technique promoted in this thesis) this strategy will not be considered further, and where appropriate, one of the three remaining strategies where one set of attributes is calculated will be chosen.

Strategy 2 Expected attributes are computed in this strategy:

$$t_1(if) = \alpha \cdot t_1(if_t) + (1 - \alpha) \cdot t_1(if_f)$$

$$t_2(if) = \alpha \cdot t_2(if_t) + (1 - \alpha) \cdot t_2(if_f)$$

$$IP(if) = \alpha \cdot IP(if_t) + (1 - \alpha) \cdot IP(if_f)$$

$$n(if) = \alpha \cdot n(if_t) + (1 - \alpha) \cdot n(if_f)$$

$$= 2 + \alpha \cdot n(then) + (1 - \alpha) \cdot n(else)$$

where the attributes of if_t and if_f are computed as in strategy 1. The main disadvantage with this approach is that the influence the conditional may have upon the period p of a stream is localized whereas it may have a more noticeable impact when taken in a more global context. For example, suppose the conditional was in a streamed computation where all the other influences other than IP(if) upon the period of the stream are represented as θ. Letting $x = IP(if_t)$ and $y = IP(if_f)$, then strategy 1 would more accurately yield

$$p = \alpha \cdot \max(x, \theta) + (1 - \alpha) \cdot \max(y, \theta)$$

where strategy 2 yields

$$p = \max[\, \theta, \alpha \cdot x + (1 - \alpha) \cdot y]$$

Though this discrepancy may be large, in the interest of

simplicity this approach will be used for conditionals not embedded within recursive streamed procedures. For recursive streamed procedures, a special technique will be introduced later.

Strategy 3 Maximum attributes are calculated:

$t_1(if) = \max[t_1(if_t), t_1(if_f)]$

$t_2(if) = \max[t_2(if_t), t_2(if_f)]$

$IP(if) = \max[IP(if_t), IP(if_f)]$

$n(if) = \max[n(if_t), n(if_f)]$

This strategy captures the "worst" influence the conditional could have on the enclosing streamed computation. This influence may be accurate in the case of the independent period when the condition is found within a streamed recursive procedure. In this situation, the stream is passed through many occurrences of the conditional and it is unlikely that it would not pass through at least one occurrence of the then body and at least one occurrence of the else body.

Strategy 4 Either the attributes of if_t or if_f are chosen to represent the attributes of the conditional. This strategy is reasonable when the condition is always true or always false or when the underlying semantics imply that one alternative occurs only when it cannot have a noticeable impact upon the computation (e.g., when the if_t is executed only in the grounded invocation of a recursive procedure).

This is illustrated in the next section dealing with a class of streamed recursive procedures.

Other strategies are possible; e.g., using minimum attributes or computing different attributes with different strategies. The suitability of applying a given approach is dependent upon the context in which the conditional construct is found. The automated recognition of this context may be a very difficult task. However, for the purposes of this investigation, their appropriate use is described without reference to the detection of their context. Conditionals not found within recursive procedures will be analyzed with strategy 2 (expected attributes). Conditionals within recursive procedures where the conditional probability, α, is not expressed in terms of the length of the stream, m, will have t_1, t_2, and n attributes calculated by strategy 2 and the IP attribute computed by strategy 3 (since there exists a high probability of multiple activations of <u>both</u> choices). Conditionals within streamed recursive procedures producing output streams where the conditional probability is expressed in terms of the length of the stream (e.g., $1/(m + 1)$) will have their attributes calculated by strategy 4 (since usually one choice only affects the grounded invocation). Conditionals with streamed recursive procedures functioning as stream sinks where the conditional probability is expressed in

terms of the length of the stream will have the t_1 attribute computed by strategy 2 and the IP attribute computed by strategy 4. This is further illustrated in the following section on procedures.

Procedures (streamed)

The analysis of a non-recursive streamed procedure is done in the same manner as described for a streamed block. The reader is referred to a previous section.

Many streamed recursive procedures take the form of processing a few tokens of the input stream(s) and creating a few tokens of the output stream(s) in each invocation while simultaneously passing the remainder of the input stream to a recursive invocation to produce the remainder of the output stream. A high degree of parallelism may result as the bodies of successive invocations overlap. A sample program appears in Figure 4.37. In this example the abstract operation f is performed on each token of the input

```
PROCEDURE P (REAL STREAM S)
      RETURNS (REAL STREAM T)
   T = IF EMPTY(S) THEN EOS
                   ELSE BEGIN
                        REAL STREAM R;
                        R = CONS{f[FIRST(S)], P[REST(S)]}
                        END (R)
   END
```

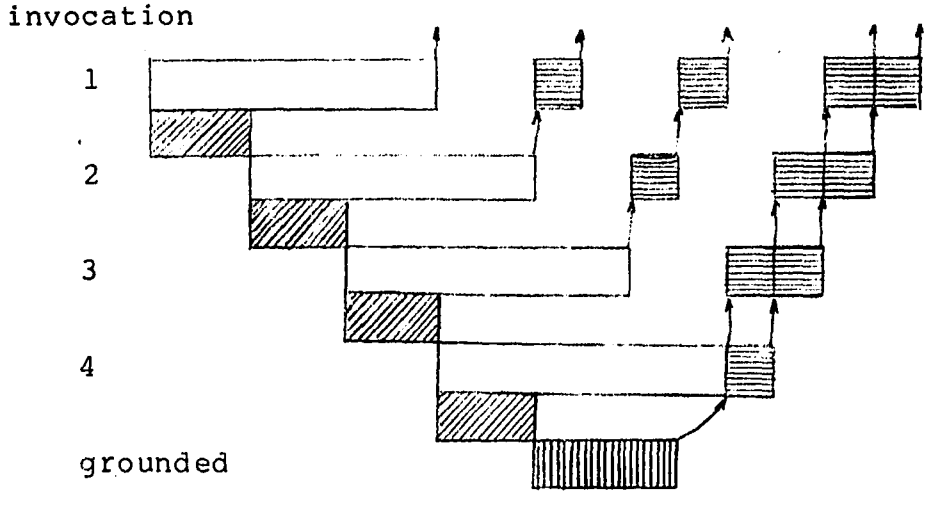
Figure 4.37. Sample streamed recursive procedure

streams to produce the output stream T. This example is just one of the many possible types of streamed recursive procedures. These procedures may function as stream source, phase, or sink nodes or any possible combination of these types. Consider the procedure of Figure 4.38 that recursively produces a stream of integers, f(i) through f(m). This procedure functions as a stream source (scalar inputs, stream output) and a timing diagram appears in Figure 4.39. For the purposes of discussion, denote the abstract operations involved as B, R, and C where the "node" B computes the output stream value produced by a given invocation, the "node" R constructs the arguments for, and makes the invocation of, the recursive call, and the "node" C processes and returns the stream values produced by recursive invocations. The Petri net representation of the procedure of Figure 4.38 appears in Figure 4.40. The dynamic code unraveling that occurs as successive invocations of Q are made is illustrated in Figure 4.41. As was mentioned in the section on

```
PROCEDURE Q (INTEGER I,M)
      RETURNS (INTEGER STREAM S)
   S = IF I. > M THEN EOS
                 ELSE BEGIN
                    INTEGER STREAM T;
                    T = CONS[f(I), Q(I+1, M)]
                    END (T)
   END
```

Figure 4.38. Recursive stream source

streamed conditionals, this illustrates a case where one of the alternate bodies (in this case, the then body--the "grounding" body) has little influence on the overall behavior of the timing of the recursive procedure. In fact, the probability of choosing the then body is $\alpha = 1/(m + 1)$



time to compute the output stream value f(i) produced by this invocation $t_1(B)$

time to initiate recursive call $t_1(R)$

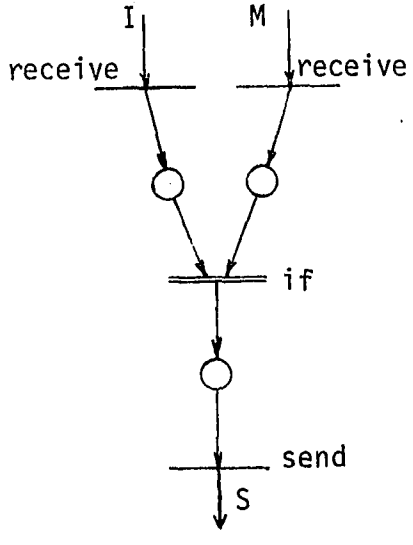
time to process and return stream values produced by recursive invocations $t_1(C)$

time of grounded invocation

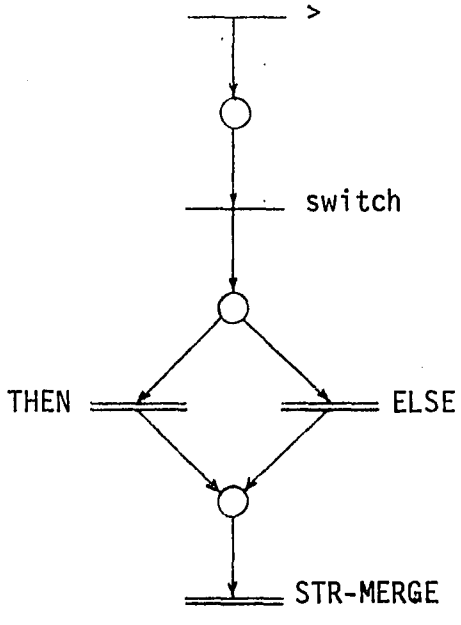
Figure 4.39. Timing diagram of a recursive procedure generating a stream

for an initial invocation of Q(1,m). α approaches zero for large values of m. In situations such as this, the attributes of the conditional will be those of if_f.

As can be seen from the timing diagram (Figure 4.39), $t_1(Q) = t_1(B)$. Following the normal pattern of nodal



(a) Procedure Q

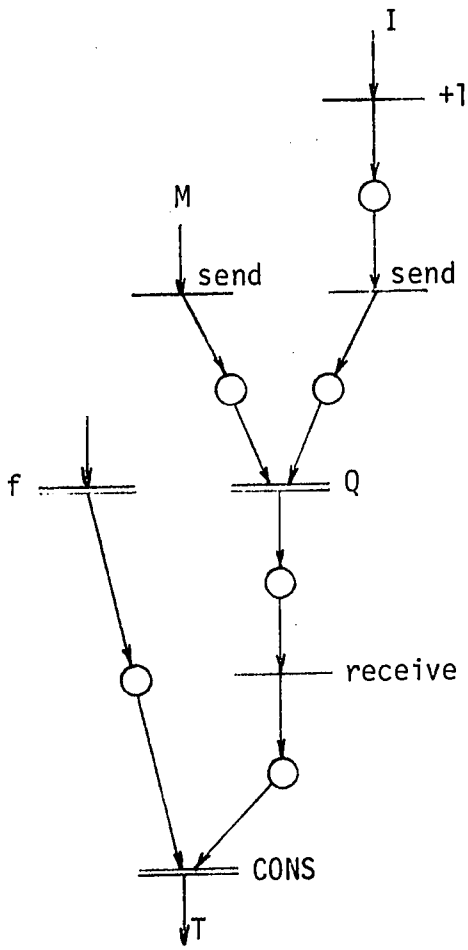


(b) IF node

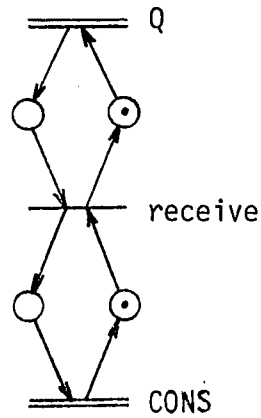
——— identity

(c) THEN node

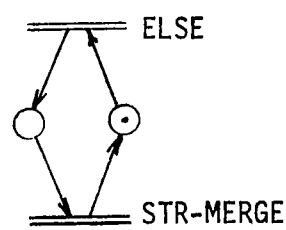
Figure 4.4Ø. Petri net representation of procedure of Figure 4.38



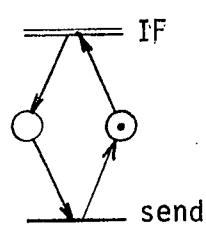
(d) ELSE node



(e) Petri net ELSEp



(f) Petri net ^{f}IFp



(g) Petri net Qp

Figure 4.40. Continued

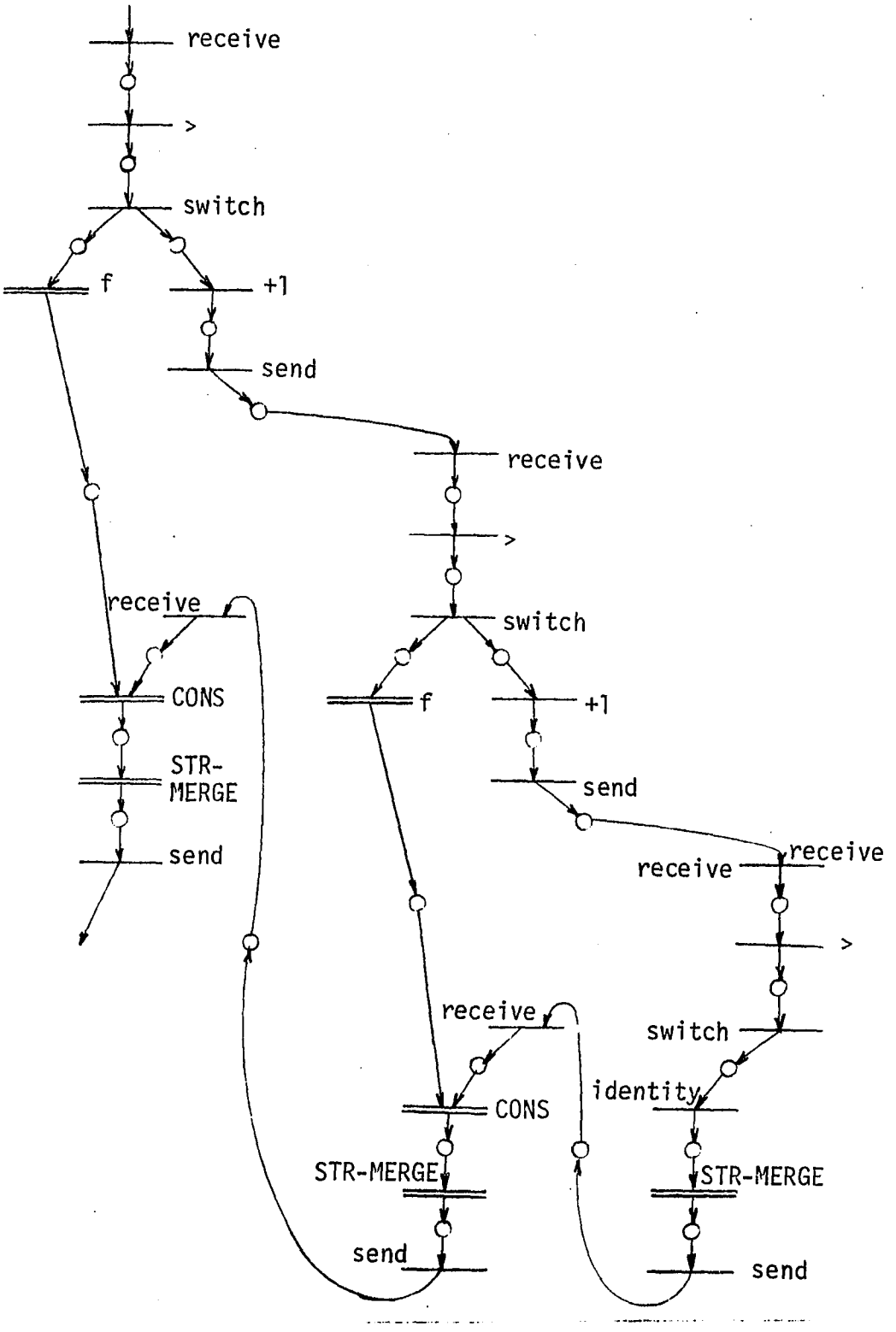


Figure 4.41. Dynamic code unraveling of recursive procedure functioning as a stream source

reductions:

$$t_1(Q) = \tau(Qs_1) = t(\text{receive}) + t_1({}^{f}IF) + t(\text{send})$$
$$= 2 + t_1({}^{f}IF)$$
$$t_1({}^{f}IF) = \tau({}^{f}IFs_1) = t(>) + t(\text{switch}) + t_1(\text{ELSE}) + t_1(\text{STR-MERGE})$$
$$= 3 + t_1(\text{ELSE})$$
$$t_1(\text{ELSE}) = \tau(\text{ELSEs}_1) = t_1(f) + t_1(\text{CONS})$$
$$= 1 + t_1(f)$$

Therefore, $t_1(Q) = 6 + t_1(f)$ as may also be seen in Figure 4.41. Two factors influence the period of the stream output from Q:

a) the normal influences as a result of the state machines of Qp and internal nodes' IP, and

b) the delay in constructing successive output tokens.

These influences are captured by a careful computation of IP(Q):

$$IP(Q) = \max[\tau(Qp), IP({}^{f}IF)]$$
$$= \max[2, IP({}^{f}IF)]$$
$$IP({}^{f}IF) = \max[\tau({}^{f}IFp), IP(\text{ELSE}), IP(\text{STR-MERGE})]$$
$$= \max[2, IP(\text{ELSE}), 3]$$

The independent period of the stream output from the ELSE node is computed as (see Equation (4.14)):

$$IP(\text{ELSE}) = (d_{1,2} + m\ p')/(m + 1)$$
$$= [\max\{IP(\text{CONS}), [t(+) + t(\text{send}) + t_1(Q) + t(\text{receive})] - t_1(f)\}$$
$$+ m \cdot \max[\tau(\text{ELSEp}), IP(Q), IP(\text{CONS})]]/(m + 1)$$

Substituting in for $t_1(Q)$ and known attributes,

$$IP(ELSE) = [\max\{2, [3 + (6 + t_1(f))] - t_1(f)\} + m \cdot \max[2, IP(Q), 2]]/(m + 1)$$
$$= \{\max(2,9) + m \cdot \max[2, IP(Q)]\}/(m + 1)$$
$$= \{9 + m \cdot \max[2,IP(Q)]\}/(m + 1)$$

Substituting this latter result into IP(IF) which in turn is substituted into IP(Q),

$$IP(Q) = \max[3, \{9 + m \cdot \max[2,IP(Q)]\}/(m + 1)]$$

Two cases result:

Case 1: IP(ELSE) <= 3

Therefore, IP(Q) = max[IP(ELSE), 3] = 3.

Case 2: IP(ELSE) > 3

Therefore, IP(Q) = {9 + m · max[IP(Q), 2]}/(m + 1).

Solving for IP(Q) and assuming large m, IP(Q) = 9.

By taking the maximum of both cases, IP(Q) = max[3,9]. Case 1 points out the controlling influence that the abstract "node" C has on the period of the output stream: any stream returned by the procedure passes through this portion of the initial invocation (the influences other invocations have in this manner are the same, though on a smaller segment of the stream). Case 2 shows the influence that the delay in making the recursive call and returning stream tokens has on the period. Abstractly this is expressed as

$$[t_1(R) + t_1(C)]/a$$

where a is the number of tokens produced by each invocation. This is pictorially illustrated by a close examination of Figure 4.39. The key in capturing this influence arose in not assuming $d_{1,2} = p$ in the else body. This is crucial since the period of this procedure is determined by the cumulative effect of the $d_{1,2}$ value of each invocation.

Returning to the original example of Figure 4.37, the procedure presented there is analyzed in much the same manner. A timing diagram appears in Figure 4.42. The dynamic code unraveling that occurs as successive invocations of P are made is illustrated in Figure 4.43. Two important points need to be made concerning the behavior of this type of procedure. First and most important, the streams S and T in a strict sense are not part of the same streamed computation; i.e., there exist only scalar connections (dotted arcs in Figure 4.43) relating tokens of S to tokens of T. This is accomplished by sinking the input stream S (in this example, via the FIRST operation) in each invocation, producing a value to be used in the construction of a new output token of stream T. As a result, a recursive procedure of this type simultaneously functions as a stream source and a stream sink for two separate stream computations. There is a high degree of overlap between these two streamed computations however, so therefore, this type of procedure will be treated as a special type of phase node.

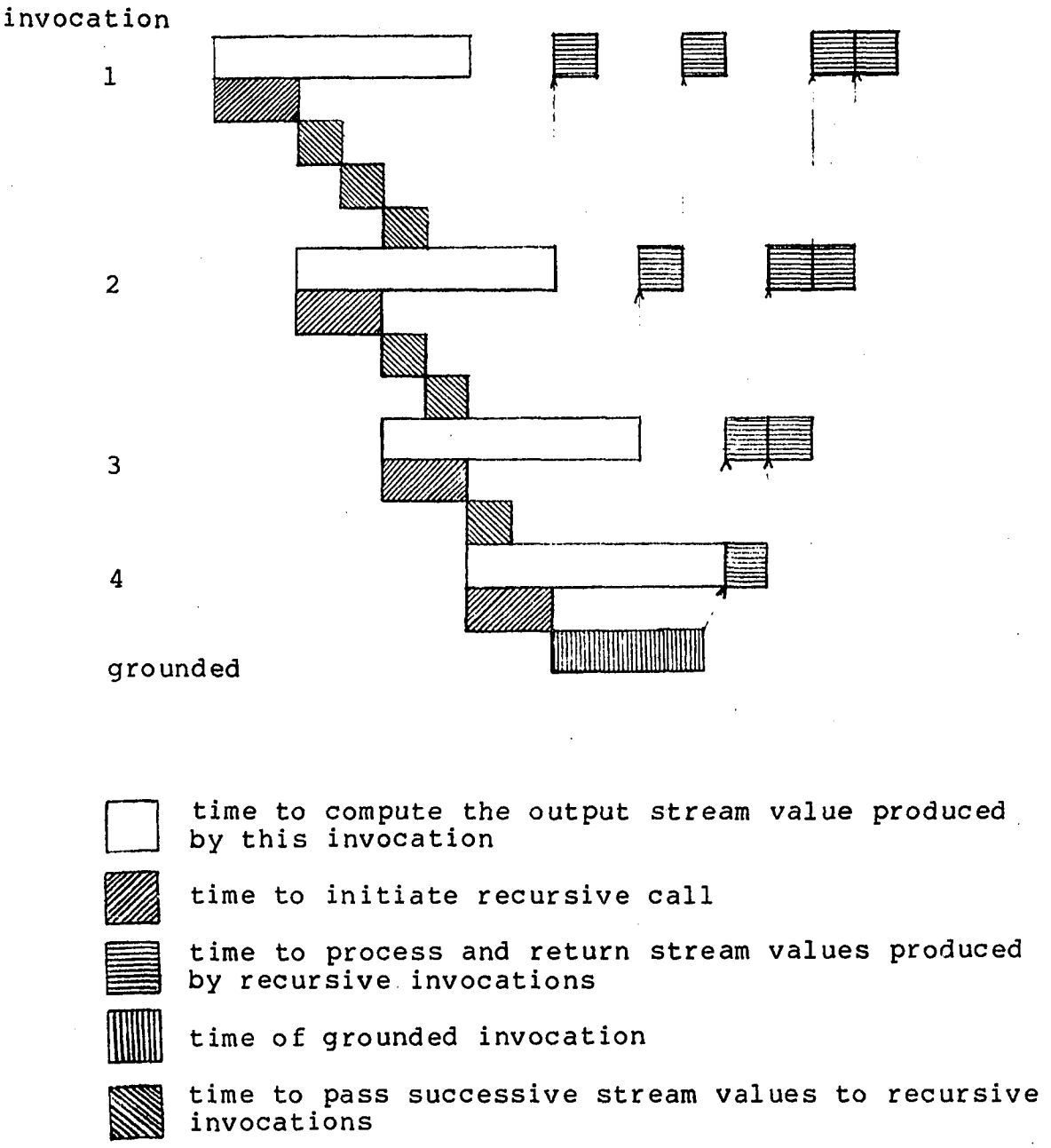


Figure 4.42. Timing diagram of recursive procedure with stream input and output parameters

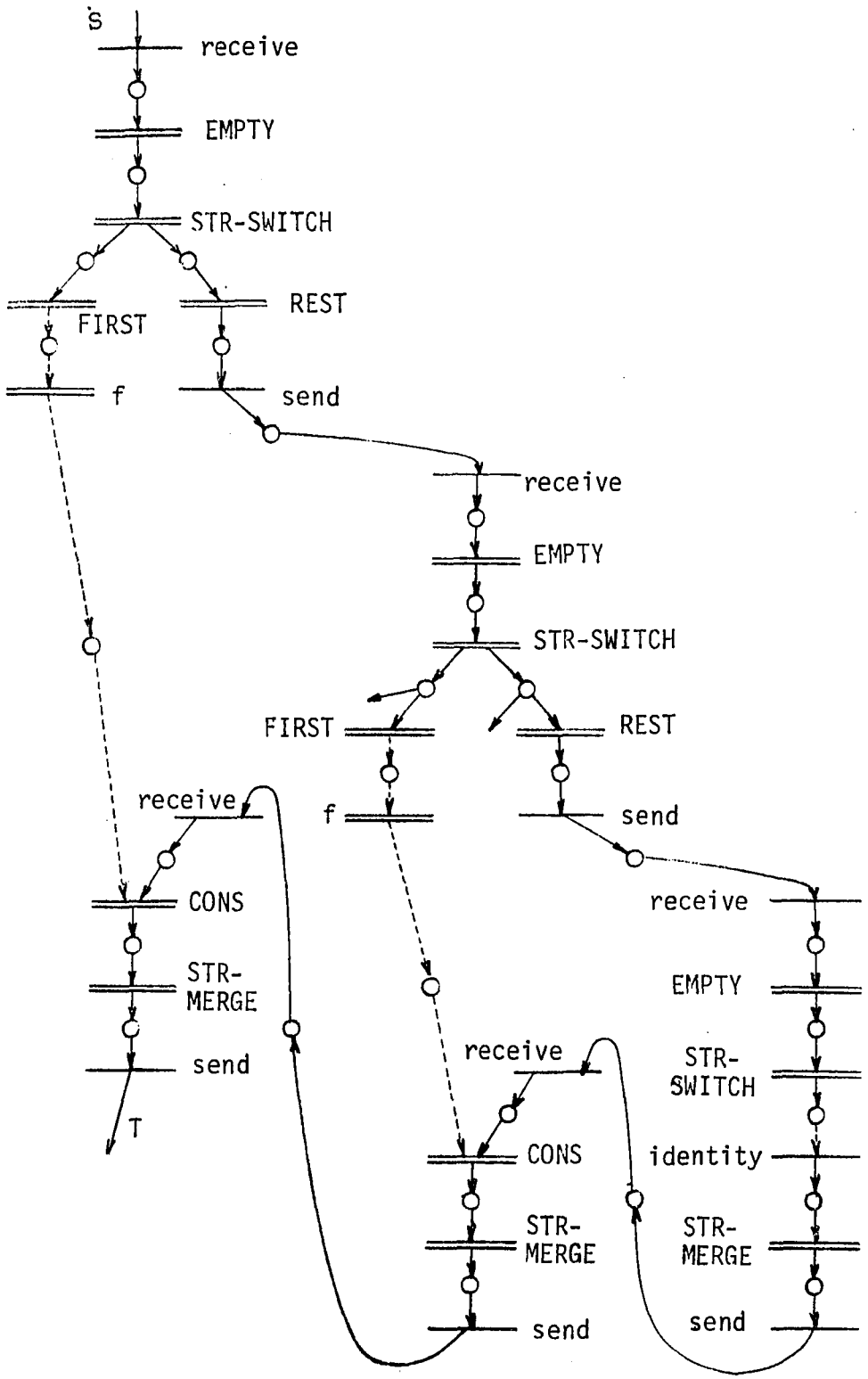


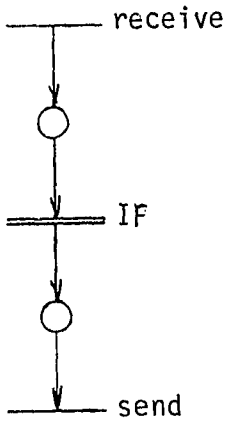
Figure 4.43. Dynamic code unraveling of recursive procedure with stream input and output parameters

The second point deals with the hierarchal ordering of nodes. Since this type of procedure includes portions of two streamed computations, the procedure node is not strictly internal to either one. Consequently, this type of procedure will be treated as partitioning the streamed computation into two parts and the internal portions of the two streamed computations will be analyzed separately when appropriate. For example, consider the Petri net representation of the node P found in Figure 4.44. As was done in the previous example, the attributes of the conditional are computed as those of if_f. Following the normal nodal reduction pattern (without reducing the internal portions of the streamed computations), the value for $t_1(P)$ is computed as

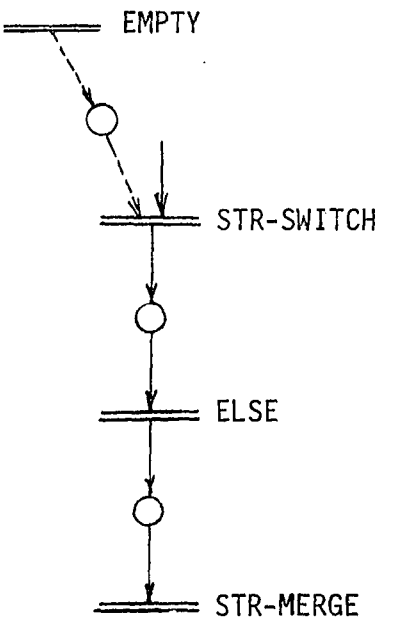
$$t_1(P) = \tau(Ps_1) = t(\text{receive}) + t_1({}^{f}\text{IF}) + t(\text{send})$$
$$= 2 + t_1({}^{f}\text{IF})$$
$$t_1({}^{f}\text{IF}) = \tau({}^{f}\text{IFs}_1) = t_1(\text{EMPTY}) + t_1(\text{STR-SWITCH}) + t_1(\text{ELSE}) + t_1(\text{STR-MERGE})$$
$$= 3 + 2 + t_1(\text{ELSE}) + 1$$
$$= 6 + t_1(\text{ELSE})$$
$$t_1(\text{ELSE}) = t_1(\text{FIRST}) + t_1(f) + 1$$
$$= 3 + t_1(f)$$

Therefore, $t_1(P) = 11 + t_1(f)$.

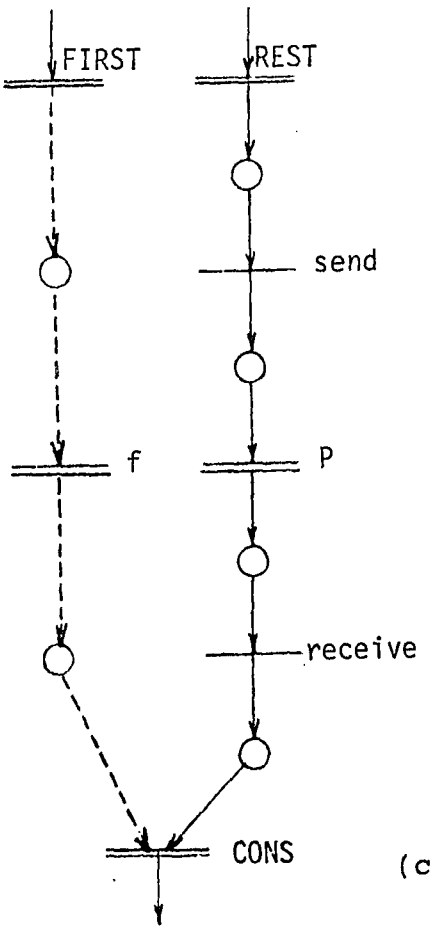
Since P functions as a part of two streamed computations, two values are computed for IP(P): $IP_{in}(P)$ and



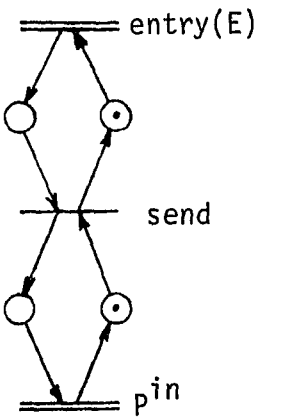
(a) Procedure P



(b) f_{IF} node

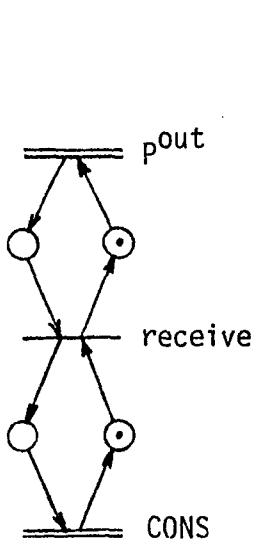


(c) ELSE node

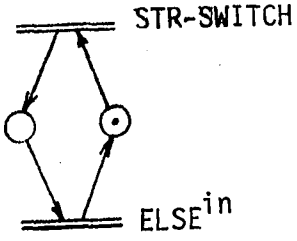


(d) Petri net ELSEp^{in}

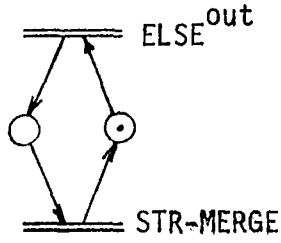
Figure 4.44. Petri net representation of procedure of Figure 4.37



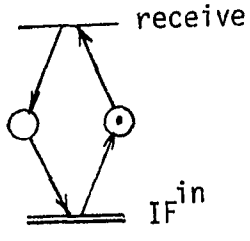
(e) Petri net ELSEp^{out}



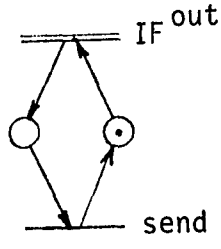
(f) Petri net $^{f}_{IFp}{}^{in}$



(g) Petri net $^{f}_{IFp}{}^{out}$



(h) Petri net Pp^{in}



(i) Petri net Pp^{out}

Figure 4.44. Continued

$IP_{out}(P)$. These values express the influence P has on the period of the two streamed computations. As is seen in Figure 4.43, $IP_{in}(P)$ involves IP(EMPTY), IP(STR-SWITCH), IP(FIRST), IP(REST), and the periods of the appropriate Np-type Petri nets. These values are computed as:

$$IP_{in}(P) = \max[\tau(Pp^{in}), IP(IF^{in})]$$
$$= \max[2, IP(IF^{in})]$$
$$IP(IF^{in}) = \max[\tau(^{f}IFp^{in}), IP(EMPTY), IP(STR\text{-}SWITCH), IP(ELSE^{in})]$$
$$= \max[2, 3, 3, IP(ELSE^{in})]$$
$$IP(ELSE^{in}) = \max[\tau(ELSEp^{in}), IP(entry), IP(P^{in})]$$
$$= \max[2, IP(entry), IP(P^{in})]$$
$$IP(entry) = \max[IP(FIRST), IP(REST)]$$
$$= \max[3,3] = 3$$
$$IP_{in}(P) = \max[3, IP(P^{in})] = 3 \qquad (4.19)$$

This definition is recursive for the m levels of recursion that P goes through until P grounds. Since the assumption that the grounded invocation does not noticeably affect the attributes of P is made, $IP_{in}(P) = 3$. Likewise,

$$IP_{out}(P) = \max[\tau(Pp^{out}), IP(IF^{out})]$$
$$= \max[2, IP(IF^{out})]$$
$$IP(IF^{out}) = \max[\tau(IFp^{out}), IP(ELSE^{out}), IP(STR\text{-}MERGE)]$$
$$= \max[2, IP(ELSE^{out}), 3]$$
$$IP(ELSE^{out}) = (d_{1,2} + m \cdot p')/(m + 1)$$
$$= [\max\{IP(CONS), [t_1(REST) + t(send) + t_1(P) + t(receive)] - [t_1(FIRST) + t_1(f)]\} + m \cdot \max[\tau(ELSEp^{out}), IP(P^{out}), IP(CONS)]] / (m + 1)$$

Substituting in for $t_1(P)$ in $IP(ELSE^{out})$ and known quantities

$$IP_{out}(P) = \max\{3, [\max\{2, [[2 + \max(3,p)] + 1 + [11 + t_1(f)] + 1] - [2 + t_1(f)]\} + m \cdot \max[2, IP(P^{out}), 2]]/(m + 1)\}$$

$$= \max\{3, [\max\{2, [13 + \max(3,p)]\} + m \cdot \max[2, IP(P^{out})]]/(m + 1)\}$$

$$= \max\{3, [13 + \max(3,p) + m \cdot \max[2, IP(P^{out})]] / (m + 1)$$

Again, two cases result,

Case 1: $IP(ELSE^{out}) <= 3$

Therefore, $IP_{out}(P) = 3$

Case 2: $IP(ELSE^{out}) > 3$

Therefore,

$$IP_{out}(P) = [13 + \max(3,p) + m \cdot IP(P^{out})]/(m + 1)$$

Since $IP(P^{out}) = IP_{out}(P)$, solving for $IP_{out}(P)$ results in

$$IP_{out}(P) = 13 + \max(3,p)$$

Taking the maximum of both cases results in

$$IP_{out}(P) = \max[3, 13 + \max(3,p)]$$

$$= 13 + \max(3,p) \qquad (4.20)$$

Since p is the period of the stream input to the REST node, p is updated to $\max[p', IP_{in}(P)]$ to reflect the influence the portion of the streamed computation internal to P may have on this period, where p' is the period of the stream input to P. Therefore,

$$IP_{out}(P) = 13 + \max(3,p') \qquad (4.21)$$

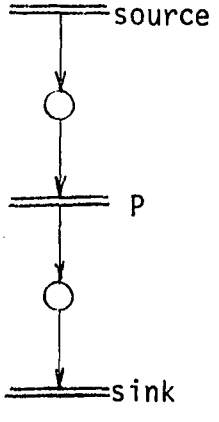
This value of $IP_{out}(P)$ reflects the dominance of the term $[t_1(R) + t_1(C)]/a$ on the period of the output stream.

Consider now how these two IP values influence their corresponding portions of the streamed computations. The simple streamed computation of Figure 4.45 uses P as a phase node. The period of the stream input to P is computed as

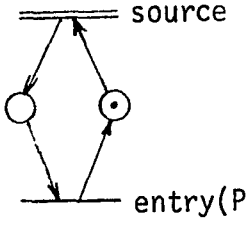
$$p_{in} = \max[\tau(Np^{in}), IP(source)]$$
$$= \max[2, IP(source)]$$

where the period of the stream input to the sink node is computed as

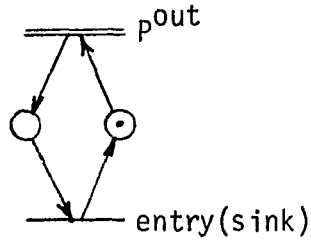
$$p_{out} = \max[\tau(Np^{out}), IP_{out}(P)]$$
$$= \max[2, IP_{out}(P)]$$



(a) Petri net N



(b) Petri net Np^{in}



(c) Petri net Np^{out}

Figure 4.45. Recursive streamed procedure as phase node

The value of $IP_{in}(P)$ was used to update the value of p in Equation (4.2Ø) to yield Equation (4.21) where p' = p_{in}. Assuming IP(source) = 4, then

$p_{in} = 4$,

$IP_{out}(P) = 13 + \max(3,4) = 17$, and

$p_{out} = \max(2,17) = 17$.

Consider now the procedure found in Figure 4.46. Since this procedure functions as a stream sink, the attributes t_2, n, and IP_{out} are not necessary. The attribute $IP_{in}(R)$ is computed in the same manner as $IP_{in}(P)$. The computation of $t_1(R)$ proceeds in the normal manner through the nodal reductions found in Figure 4.47:

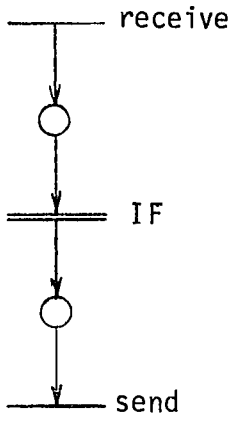
$$t_1(R) = t(\text{receive}) + t_1(IF) + t(\text{send})$$

$$= 2 + t_1(IF)$$

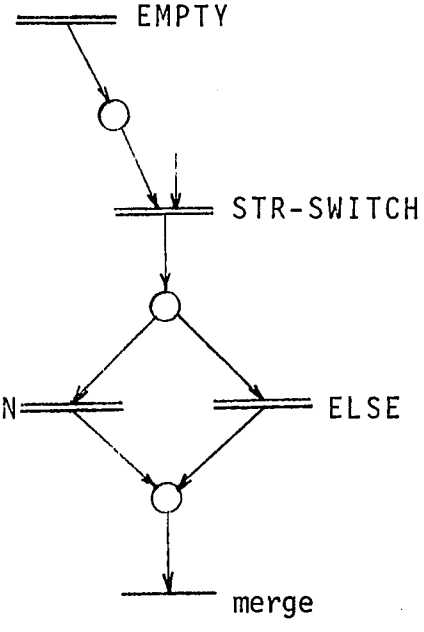
$$t_1(IF) = \alpha \cdot t_1(^{t}IF) + (1 - \alpha) \cdot t_1(^{f}IF)$$

```
PROCEDURE R (REAL STREAM S)
      RETURNS (REAL X)
   X = IF EMPTY(S) THEN Ø.Ø
                   ELSE BEGIN
                        REAL Y;
                        Y = f[FIRST(S)] + R[REST(S)]
                        END (Y)
   END
```

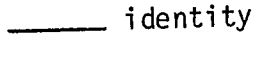
Figure 4.46. Recursive stream sink



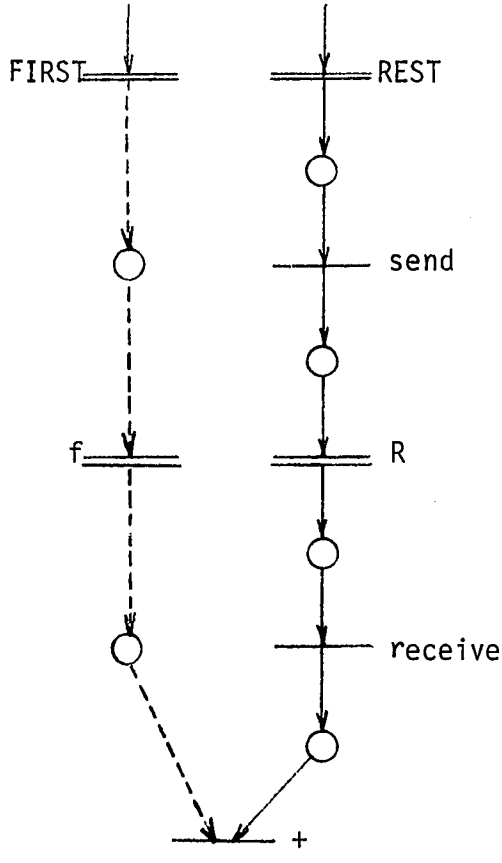
(a) Procedure R



(b) IF node



(c) THEN node



(d) ELSE node

Figure 4.47. Petri net representation of procedure of Figure 4.46

$$t_1(^{t}IF) = t_1(EMPTY) + t_1(STR\text{-}SWITCH) + t_1(THEN) + t(merge)$$
$$= 3 + 2 + t_1(THEN) + 1$$
$$= 6 + t_1(THEN)$$

$$t_1(THEN) = t(identity) = 1$$

$$t_1(^{f}IF) = t_1(EMPTY) + t_1(STR\text{-}SWITCH) + t_1(ELSE) + t(merge)$$
$$= 3 + 2 + t_1(ELSE) + 1$$
$$= 6 + t_1(ELSE)$$

$$t_1(ELSE) = \max[t_1(FIRST) + t_1(f) + t(+),\ t_1(REST) + t(send) + t_1(R) + t(receive) + t(+)]$$
$$= \max[3 + t_1(f),\ 5 + \max(3,p) + t_1(R)]$$

Since the second term will always be larger than the first (except possibly when $t_1(R)$ represents the grounded invocation),

$$t_1(ELSE) = 5 + \max(3,p) + t_1(R)$$

The p term is updated by $\max[p', IP_{in}(R)]$ where $IP_{in}(R) = 3$; therefore,

$$t_1(ELSE) = 5 + \max(3,p') + t_1(R).$$

Substituting yields

$$t_1(R) = 2 + 7(\alpha) + (1 - \alpha)\cdot[11 + \max(3,p') + t_1(R)]$$
$$= 2/\alpha + 7 + ((1 - \alpha)/\alpha)\cdot[11 + \max(3,p')]$$

Letting $\alpha = 1/(m + 1)$ and assuming $p' = 4$

$$t_1(R) = (m + 1)\cdot 2 + 7 + m \cdot [11 + \max(3,4)]$$
$$= 17(m) + 9$$

The analysis of the three previous types of procedures was very dependent upon the recognizability of the context in which the conditionals were found. Other types of streamed recursive procedures are handled using different strategies. For example, the innermost conditional of the procedure of Figure 4.48 has a t_1 attribute computed as the expected value of the t_1 attributes of the bodies and an IP value computed as the maximum of the IP values of the bodies. The reasoning for the t_1 attribute to be so computed is that it is not determinate which body will produce the first output token. Since after many invocations it is unlikely that the input stream is not passing through at least one occurrence of the then body and at least one

```
PROCEDURE X (INTEGER STREAM S)
      RETURNS (INTEGER STREAM T)
   T = IF EMPTY(S) THEN EOS
         ELSE BEGIN
              INTEGER STREAM U;
              U = IF EVEN(FIRST(S))
                     THEN BEGIN
                          INTEGER STREAM V;
                          V = CONS{f[FIRST(S)], X[REST(S)]}
                          END (V)
                     ELSE BEGIN
                          INTEGER STREAM W;
                          W = CONS{g[FIRST(S)], X[REST(S)]}
                          END (W)
              END (U)
   END
```

Figure 4.48. Conditionals internal to recursive procedure

occurrence of the else body, the maximum IP value is computed. Another example appears in Figure 4.49. This procedure upon being invoked by Y(S,1) produces an output stream T such that:

$T_1 = 1$

$T_2 = 2$

$T_i = (S_{i-2})^2$ for $3 <= i <= m$

The delays, $d_{1,2}$ and $d_{2,3}$, here play a negligible role in the overall period but the input stream still passes through two invocations of the then body so the appropriate computation of an IP(Y) takes the maximum value of the IP's of both bodies. The appropriate assumption that $d_{1,2} = d_{2,3} = p$ is based on a large input stream. Since these values in other procedures prove to be crucial (see procedures P and Q of Figures 4.37 and 4.38), the detection of the context in

```
PROCEDURE Y (INTEGER STREAM S, INTEGER I)
     RETURNS (INTEGER STREAM T)
  T = IF I < 3 THEN BEGIN
                    INTEGER STREAM U;
                    U = CONS[I, Y(S, I+1)]
                    END (U)
               ELSE BEGIN
                    INTEGER STREAM V;
                    V = S * S
                    END (V)
  END
```

Figure 4.49. Procedure prefixing and processing input stream

which the conditional and CONS constructs are found is further underlined.

Since in the feedback model procedures are assumed to have individual ports for each parameter, it is possible to construct examples where the assumption that all input (output) parameters are simultaneously available (released) is violated by the actual execution time behavior of the procedure (this may also be done in other constructs). Such an example appears in Figure 4.5Ø where the analysis assumes that the first token of REST(S) and X must be available before a recursive invocation of Z may be made. In actuality, a high degree of unraveling may occur in passing the stream to successive invocations. For the most part it is believed that these violations are relatively infrequent.

```
PROCEDURE Z (REAL STREAM S, REAL W)
      RETURNS (REAL STREAM T)
   T = IF EMPTY (S) THEN EOS
         ELSE BEGIN
              REAL  X,Y;
              REAL STREAM U;
              X = SUM(S);
              Y = FIRST(S)/W;
              U = CONS[Y, Z(REST(S), X)]
              END (U)
   END
```

Figure 4.5Ø. Procedure with high degree of undetected unraveling

CHAPTER V. EXPERIMENTAL RESULTS

This chapter presents the analysis and numerical results of the simulated execution [Oldehoeft, et al. 1978] of five programs for the purpose of validating the presented methodology.

Several features of this system are pertinent to mention here. Main procedures are presented with a START macro insruction. This template includes the low level instructions which trigger the release of relevant scalars that serve to enable the first executable instructions which require input data values. Procedures, as soon as it is known that they will be executed, are set up by a "call" macro that brings in the procedure and establishes the appropriate input and output ports for all parameters. This macro must be completed before a send instruction is enabled. The send instruction supports the linkage between actual arguments and formal parameters. This instruction serves to release relevant scalars which are needed in the execution of the procedure.

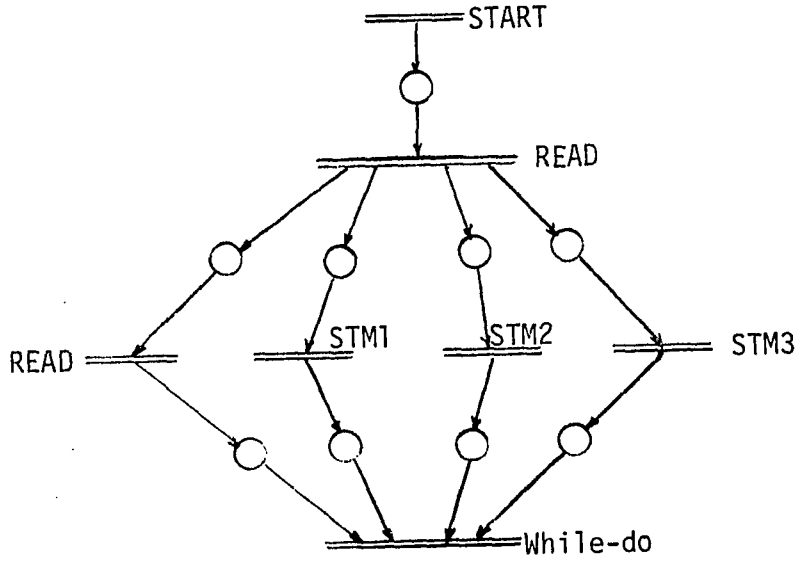
HYSL

This program is a "stripped-down" model of a three-dimensional hydrodynamic program. The high level code for this program appears in Figure 5.1. It consists primarily of a while-do whose body is a streamed computation.

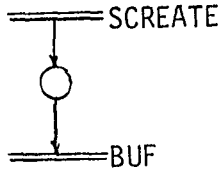
```
PROCEDURE HYSL
   REAL ARRAY A,B,C;
   FILE INFILE
   INTEGER maxz, endcycle;

   INPUT maxz FROM FILE1;        (* read *)
   INPUT endcycle FROM FILE1;    (* read *)

   A,B,C =
         WHILE TS2 < ENDCYCLE INITIAL
            AP = BUF[SCREATE(1,maxz,1)],     (* STM1 *)
            ARHO = BUF[SCREATE(1,maxz,1)],   (* STM2 *)
            AX =BUF[SCREATE(1,maxz+1,1)],    (* STM3 *)
            TS2 = Ø DO
         INTEGER TS2;
         REAL ARRAY AP, ARHO, AX;
         REAL STREAM P, RHO, X, ALL, Q, NP, NRHO, NX;

         P = UNBUF(AP);  (* Beginning of STM *)
         RHO = UNBUF(ARHO);
         X = UNBUF(AX);
         ACC = CONS[2 * FIRST(P) / FIRST(RHO),
                  REPL(2,P) * P/RHO];
         NX = X + ACC * REPL(2, ACC);
         NRHO = RHO * [SUBSTM(X, 1, MAXZ) -
                    SUBSTM(X, Ø, MAXZ)];
         Q = RHO * REPL(Ø.5, RHO);
         NP = SELECT[REPL(Ø.Ø, Q), Q,
              ABS(Q) < REPL(Ø.Ø, Q)];
         AP = BUF(NP);
         ARHO = BUF(NRHO);
         AX = BUF(NX);  (* End of STM *)
         TS2 = TS2 + 1;
         END (AP, ARHO, AX)

   END
```

Figure 5.1. Program HYSL

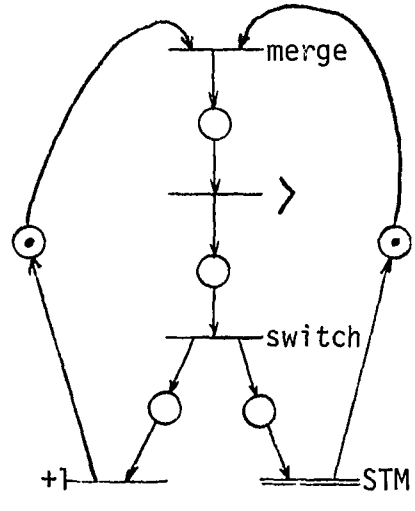
Since the while-do is a scalar operation, streams input to and output from this construct are buffered. The pertinent nodal representations appear in Figure 5.2, and the calculation of the attribute t_1(HYSL) proceeds as



(a) Procedure HYSL

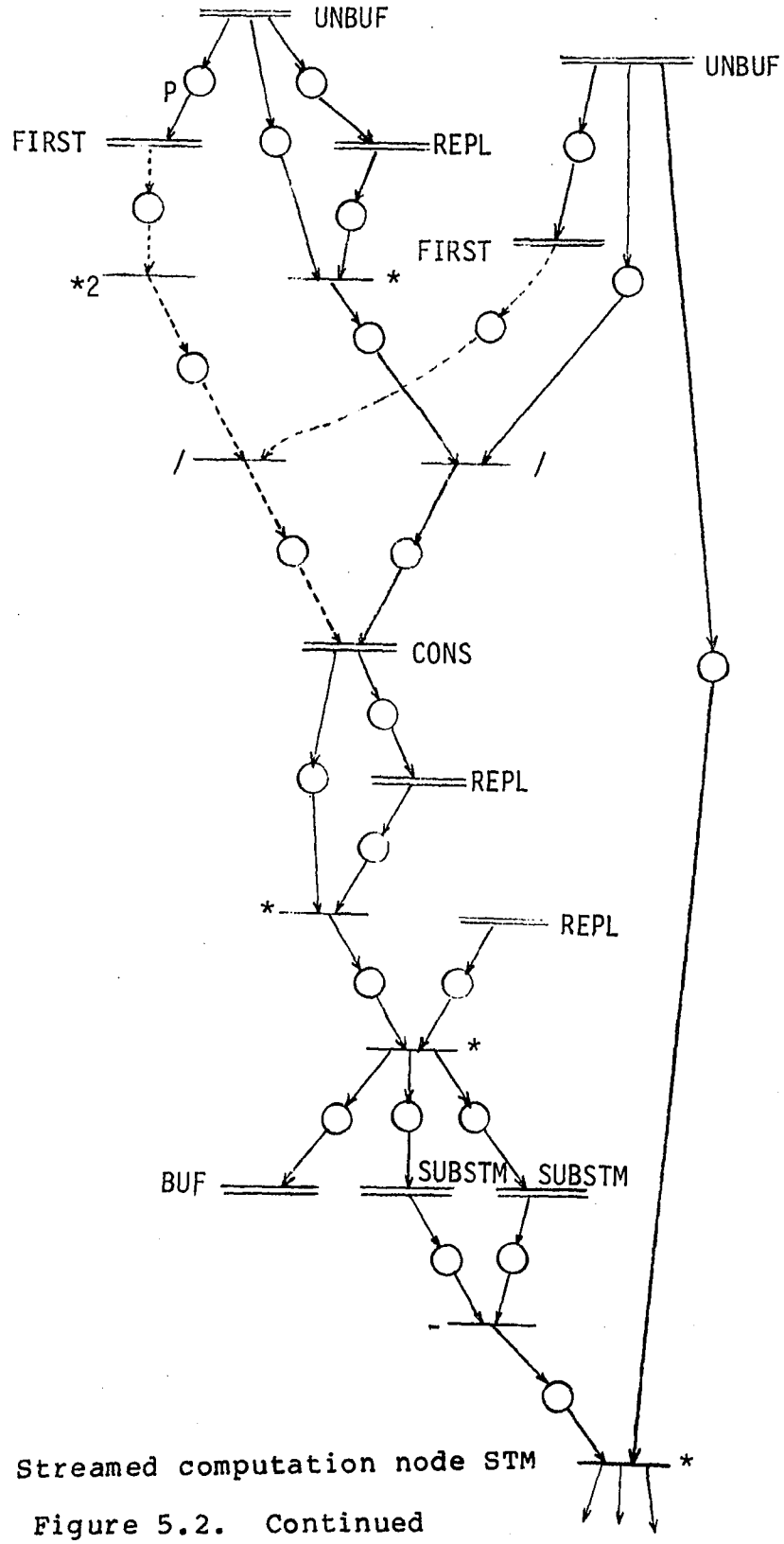


(b) Streamed Computation STM1, STM2, STM3



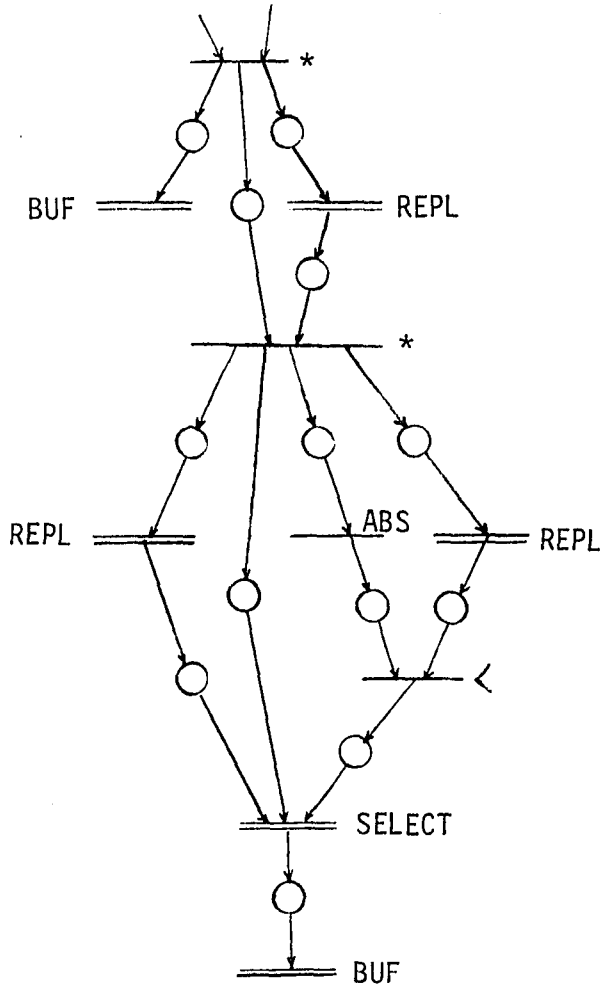
(c) Petri net W

Figure 5.2. Petri net nodes of HYSL



(d) Streamed computation node STM

Figure 5.2. Continued



(d) Continued

Figure 5.2. Continued

$$t_1(HYSL) = t_1(START) + t_1(READ) + \max[t_1(STM1), t_1(STM2), t_1(STM3), t_1(READ)] + t_1(WHILE)$$

$$t_1(STM1) = t_1(STM2) = t_1(SCREATE) + t_1(BUF)$$
$$= 5 + [3 + m \cdot \max(4,p)]$$

where

$$p = \max[\tau(STM1p), IP(SCREATE)]$$
$$= \max[2,4] = 4$$

Therefore,

$$t_1(STM1) = t_1(STM2) = 8 + 4 \quad maxz$$

The calculation of $t_1(STM3)$ proceeds similarly where $m = maxz + 1$:

$$t_1(STM3) = 8 + 4(maxz + 1)$$

The calculation of $t_1(WHILE)$ proceeds as

$$t_1(WHILE) = endcycle \cdot \tau(W) + \tau(F)$$
$$= endcycle \cdot [t(merge) + t(>) + t(switch) + t_1(STM)] + [t(merge) + t(>) + t(switch)]$$
$$= endcycle \cdot [3 + t_1(STM)] + 3$$

The dominant state machine of STM yields

$$t_1(STM) = t_1(UNBUF) + t_1(FIRST) + t(*) + t(/) + t_1(CONS) + t_1(REPL) + t(*) + t(+) + t_1(SUBSTM) + t(-) + t(*) + t_1(REPL) + t(*) + t_1(REPL) + t(<) + t_1(SELECT) + t_1(BUF)$$
$$= 5 + 2 + 1 + 1 + 1 + 4 + 1 + 1 + [7 + k \cdot \max(3,p)] + 1 + 1 + 4 + 1 + 4 + 1 + 6 + [3 + m \cdot \max(4,p)]$$
$$= 44 + k \cdot \max(3,p) + m \cdot \max(4,p)$$

The parameter k has a value 1 and p is computed as

$$p = \max[\tau(STMp), IP(UNBUF), IP(FIRST), IP(REPL), IP(CONS), IP(SUBSTM), IP(SELECT), IP(BUF)]$$

$$= \tau(STMp)$$

$$= [exit(UNBUF) + t(/) + t_2(CONS) + t_2(REPL) + t(*) + t(+) + t(SUBSTM) + t(-) + t(*)] / 1$$

$$= (1 + 1 + 1 + 4 + 1 + 1 + 5 + 1 + 1) / 1 = 16$$

This period is determined by a dominant state machine of STMp. This allows the following computations:

$$t_1(STM) = 44 + \max(3,16) + maxz \quad \max(4,16)$$

$$= 60 + 16(maxz)$$

$$t_1(WHILE) = endcycle \cdot [3 + 60 + 16(maxz)] + 3$$

$$t_1(HYSL) = 2 + 2 + t_1(STM3) + t(WHILE)$$

$$= 4 + [8 + 4(maxz + 1)] + endcycle \cdot [63 + 16(maxz)] + 3$$

$$= 19 + 4(maxz) + endcycle \quad [63 + 16(maxz)]$$

Figure 5.3 shows the simulation results for four runs.

RUN	maxz	endcycle	time steps
1	2	1	106
2	3	1	126
3	3	2	221
4	2	2	185

Figure 5.3. Simulation runs for HYSL

Solving a set of simultaneous equations yields an actual run time of

$$t_1^{smlate}(HYSL) = 19 + 4(maxz) + endcycle\ [47 + 16(maxz)]$$

The discrepancy is attributable to several factors influencing the constant term of the streamed computation:

1) scalar inputs to certain high level stream operations may arrive before the first stream token, thereby allowing reduction of firing time,
2) $d_{1,2} = 6$ for the SUBSTM operation and not the value of $p = 16$, and
3) $d_{last,eos} < p = 16$.

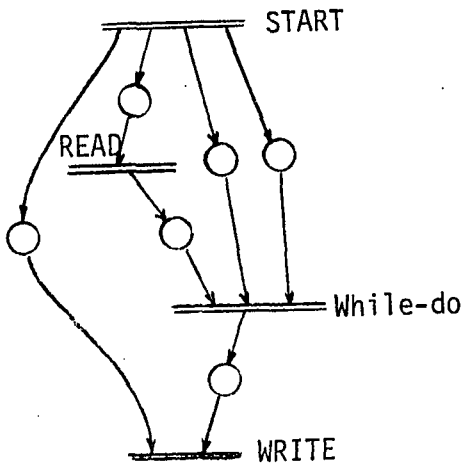
NEXP

This program computes a value for $\Sigma_i(e^{x_i} + e^{-x_i})/2$ for an input sequence x. It consists primarily of a procedure invocation internal to a while body. Internal to the procedure is a streamed computation with a recursive procedure functioning as a stream source. The high level code appears in Figure 5.4 with pertinent nodal representations in Figure 5.5. The calculation of attributes follows:

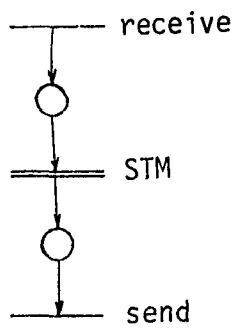
$$t_1(NEXP) = \tau(NEXPs_1)$$
$$= t_1(START) + t_1(READ) + t_1(WHILE) + t_1(WRITE)$$
$$= 2 + 2 + t_1(WHILE) + 1 = 5 + t_1(WHILE)$$

```
PROCEDURE NEXP
   INTEGER n;
   REAL S;
   FILE INFILE, OUTFILE;
   PROCEDURE EXP (REAL X) RETURNS (REAL E)
      E = STREAMED
          INTEGER STREAM P,F;
          REAL E;
          PROCEDURE FACT (INTEGER I, PREV, N)
                RETURNS (INTEGER STREAM F)
             F = IF I <= N THEN EOS
                    ELSE BEGIN
                    INTEGER Y;
                    INTEGER STREAM F;
                    Y = PREV * I;
                    F = CONS[X, FACT(X, I+1, N)]
                    END (F)
             END;
          F = FACT(1,1,3)
          P = SCREATE(1,3,1);
          E = SUM(REPL(X,P) ^ P / F)
          END (E)

   INPUT n FROM INFILE;
   S = WHILE I <= n INITIAL I = 1, S = 0.0 DO
          INTEGER I;
          REAL S,E,X;
          INPUT X FROM INFILE;
          E = EXP(X)
          S = S + (E + 1./E) / 2.;
          I = I + 1
          END (S);
   OUTPUT S TO OUTFILE
   END
```

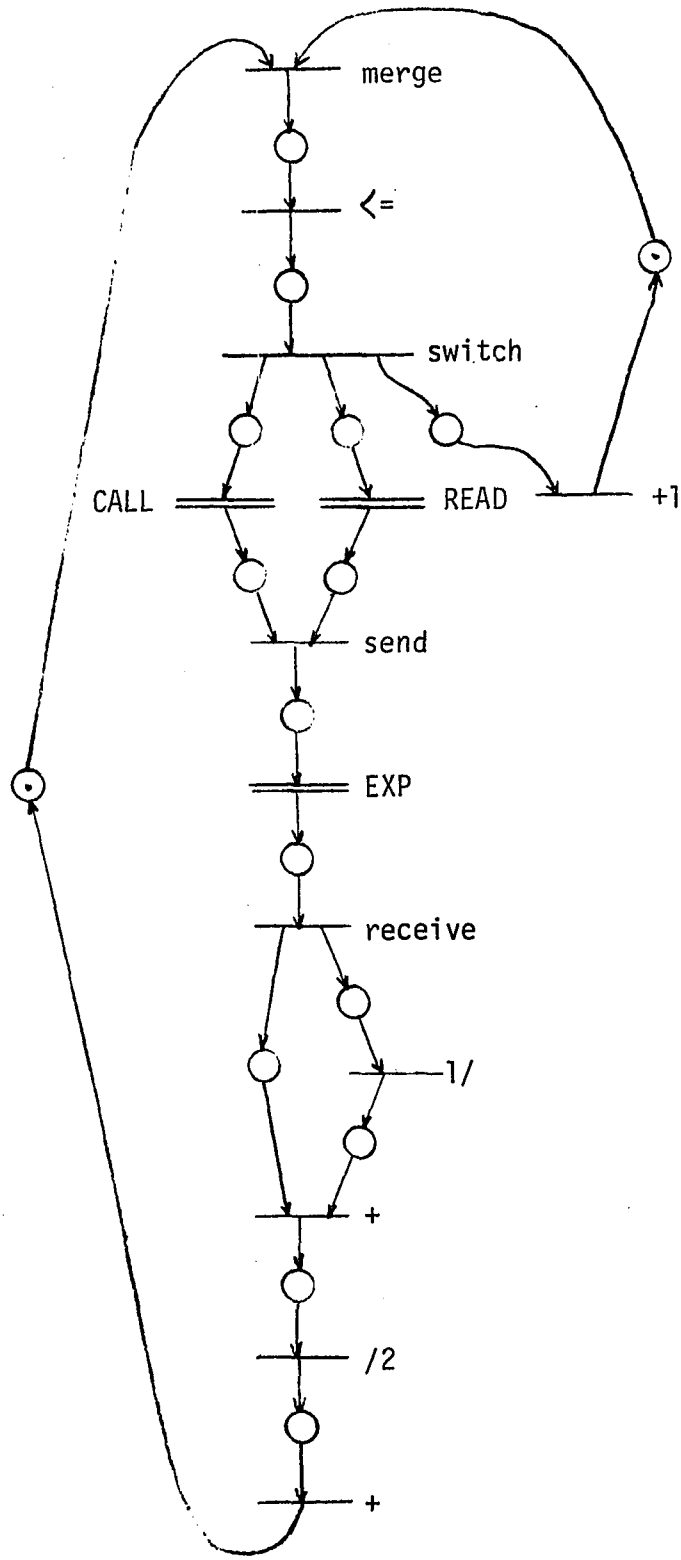
Figure 5.4. Program NEXP



(a) Procedure NEXP

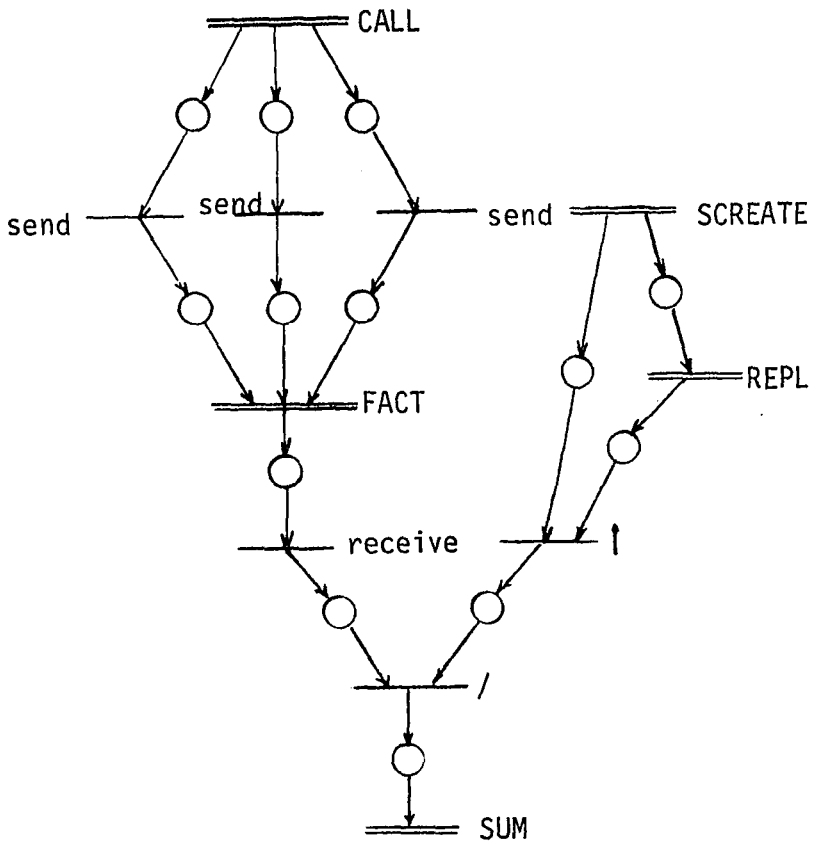


(c) Procedure EXP

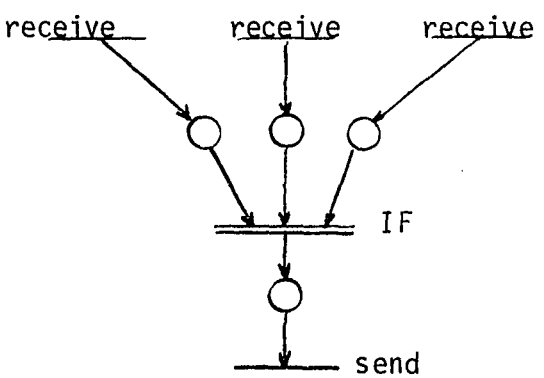


(b) Petri net W

Figure 5.5. Petri net nodes of NEXP

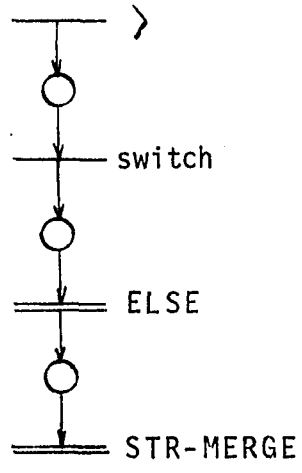


(d) Streamed computation node STM

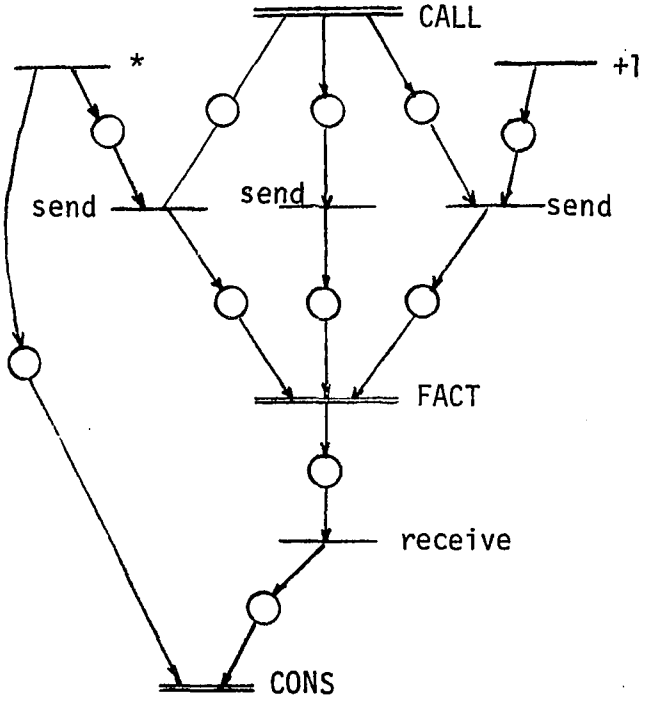


(e) Procedure FACT

Figure 5.5. Continued



(f) f_{IF} node



(g) ELSE node

Figure 5.5. Continued

$$t_1(\text{while}) = n \cdot \tau(W) + \tau(F)$$
$$= n \cdot [t(\text{merge}) + t(<=) + t(\text{switch}) + \max[t_1(\text{CALL}), t_1(\text{read})] + t(\text{send}) + t_1(\text{EXP}) + t(\text{receive}) + t(/) + t(+) + t(/) + t(+)] + [t(\text{merge}) + t(<=) + t(\text{switch})]$$
$$= n \cdot [1 + 1 + 1 + \max(4,2) + 1 + t_1(\text{EXP}) + 1 + 1 + 1 + 1 + 1] + [1 + 1 + 1]$$
$$= n \cdot [13 + t_1(\text{EXP})] + 3$$

$$t_1(\text{EXP}) = t(\text{receive}) + t_1(\text{STM}) + t(\text{send})$$
$$= 1 + t_1(\text{STM}) + 1 = 2 + t_1(\text{STM})$$

$$t_1(\text{STM}) = t_1(\text{CALL}) + t(\text{send}) + t_1(\text{FACT}) + t(\text{receive}) + t(/) + t_1(\text{SUM})$$
$$= 4 + 1 + t_1(\text{FACT}) + 1 + 1 + 2 + 3 \cdot \max(3,p)$$
$$= 9 + t_1(\text{FACT}) + 3 \cdot \max(3,p)$$

$$t_1(\text{FACT}) = t(\text{receive}) + t_1(\text{if}) + t(\text{send})$$
$$= 1 + t_1({}^{f}\text{IF}) + 1 = 2 + t_1({}^{f}\text{IF})$$

$$t({}^{f}\text{IF}) = \tau({}^{f}\text{IFs}_1) = t(>) + t(\text{switch}) + t_1(\text{ELSE}) + t_1(\text{STR-MERGE})$$
$$= 1 + 1 + t_1(\text{ELSE}) + 1 = 3 + t_1(\text{ELSE})$$

$$t_1(\text{ELSE}) = t(*) + t_1(\text{CONS}) = 1 + 1 = 2$$

Therefore,

$$t_1(\text{STM}) = 16 + 3 \max(3,p)$$

where p is computed as

$$p = \max[\tau(\text{STMp}), IP(\text{FACT}), IP(\text{SCREATE}), IP(\text{REPL})]$$
$$= \max[6, IP(\text{FACT}), 4, 3]$$

IP(FACT) is dominated by the delay in making a recursive invocation and returning successive stream values and is computed as

$$IP(FACT) = [t(receive) + t(>) + t(switch) + t_1(CALL) + t(send)] + [t(receive + t_1(CONS) + t(merge) + t(send)]$$
$$= 12$$

Therefore,

$$p = 12$$

and

$$t_1(STM) = 16 + 3 \quad max(3,12) = 16 + 3(12) = 52$$

$$t_1(EXP) = 54$$

$$t_1(WHILE) = n \quad [13 + 54] + 3$$
$$= 67(n) + 3$$

$$t_1(NEXP) = 67(n) + 8$$

Figure 5.6 shows the simulation results for two runs. Solving in terms of actual run data yields:

$$t_1^{smlate}(NEXP) = 66(n) + 8$$

This discrepancy is attributable to the fact that $d_{last,eos} < p = 12$ in the streamed computation.

RUN	n	time steps
1	1	74
2	2	140

Figure 5.6. Simulation runs for NEXP

TRIG

This program computes a value for

$$\Sigma_i \{[\sin(A_i) + \cos(A_i)]/2i\}$$

for an input stream A. It consists primarily of a streamed computation with a recursive procedure functioning as a stream sink. The high level code appears in Figure 5.7 and pertinent nodal representations appear in Figure 5.8. Attribute computations follow:

$$t_1(\text{TRIG}) = t_1(\text{START}) + t_1(\text{STM}) + t_1(\text{WRITE})$$
$$= 2 + t_1(\text{STM}) + 1 = 3 + t_1(\text{STM})$$

$$t_1(\text{STM}) = t_1(\text{STR-INPUT}) + t_1(\text{REPL}) + t(/) + t(\text{send}) + t_1(\text{SINK}) + t(\text{receive})$$
$$= 3 + 4 + 1 + 1 + t_1(\text{SINK}) + 1$$
$$= 10 + t_1(\text{SINK})$$

$$t_1(\text{SINK}) = t(\text{receive}) + t_1(\text{IF}) + t(\text{send})$$
$$= 1 + t_1(\text{IF}) + 1 = 2 + t_1(\text{IF})$$

$$t_1(\text{IF}) = t_1(\text{EMPTY}) + t_1(\text{STR-SWITCH}) + \alpha \cdot t_1(\text{THEN}) + (1 - \alpha) \cdot t_1(\text{ELSE}) + t(\text{merge})$$
$$= 3 + 2 + \alpha \cdot t_1(\text{THEN}) + (1 - \alpha) \cdot t_1(\text{ELSE})$$

$$t_1(\text{THEN}) = t(\text{constant}) = 1$$

$$t_1(\text{ELSE}) = \max[t_1(\text{FIRST}) + t(/) + t(\uparrow) + t(+), t_1(\text{REST}) + t(\text{send}) + t_1(\text{SINK}) + t(\text{receive}) + t(+)]$$
$$= [2 + \max(3,p)] + 1 + t_1(\text{SINK}) + 1 + 1$$
$$= 5 + \max(3,p) + t_1(\text{SINK})$$

```
PROCEDURE TRIG
   FILE INFILE,OUTFILE;
   REAL Z;

   Z = STREAMED
          REAL STREAM A,B;
          REAL U;
          PROCEDURE SINK (REAL STREAM X, INTEGER I)
                RETURNS (REAL V)
             V = IF EMPTY(X) THEN Ø.Ø
                    ELSE BEGIN
                       REAL W,V;
                       W = [FIRST(X) / I] ^ 2;
                       V = W + SINK[REST(X), I + 1]
                       END (V)
             END

          STR-INPUT A FROM INFILE;
          B = [SIN(A) + COS(A)] / REPL(2,A)
          U = SINK(B,1)
          END (U)
   OUTPUT Z TO OUTFILE

   END
```

Figure 5.7. Program TRIG

Substituting in, then

$$t_1(\text{IF}) = 6 + \alpha \cdot (1) + (1 - \alpha) \cdot [5 + \max(3,p) + t_1(\text{SINK})]$$

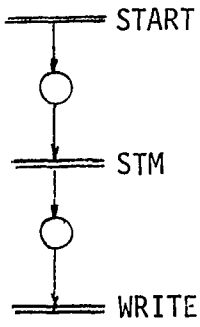
$$t_1(\text{SINK}) = 2 + 6 + \alpha + (1 - \alpha) \cdot [5 + \max(3,p) + (1 - \alpha) \cdot t_1(\text{SINK})$$

$$= 8/\alpha + 1 + [(1 - \alpha)/\alpha] \cdot [5 + \max(3,p)]$$

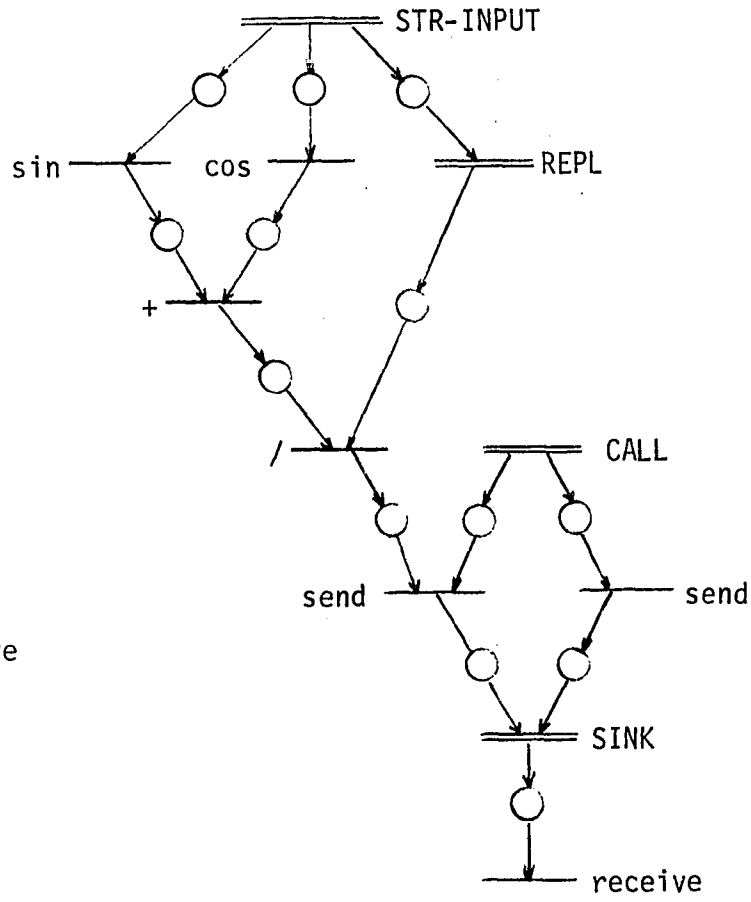
The value of p is computed as

$$p = \max[\tau(\text{STMp}), \text{IP}(\text{STR-INPUT}), \text{IP}(\text{REPL})]$$

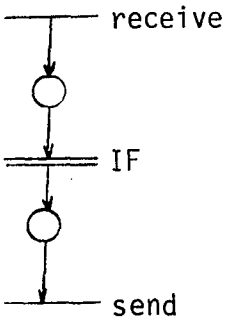
$$= \max[2,5,3] = 5$$



(a) Procedure TRIG

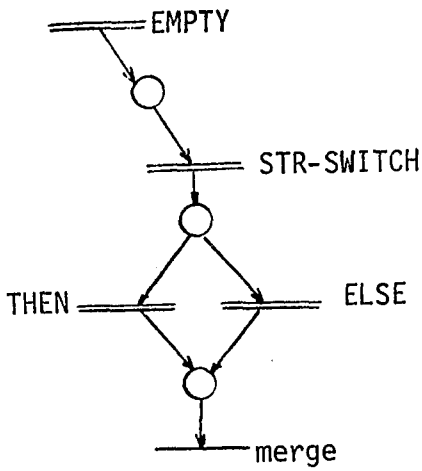


(b) Streamed computation node STM

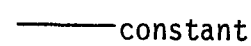


(c) Procedure SINK

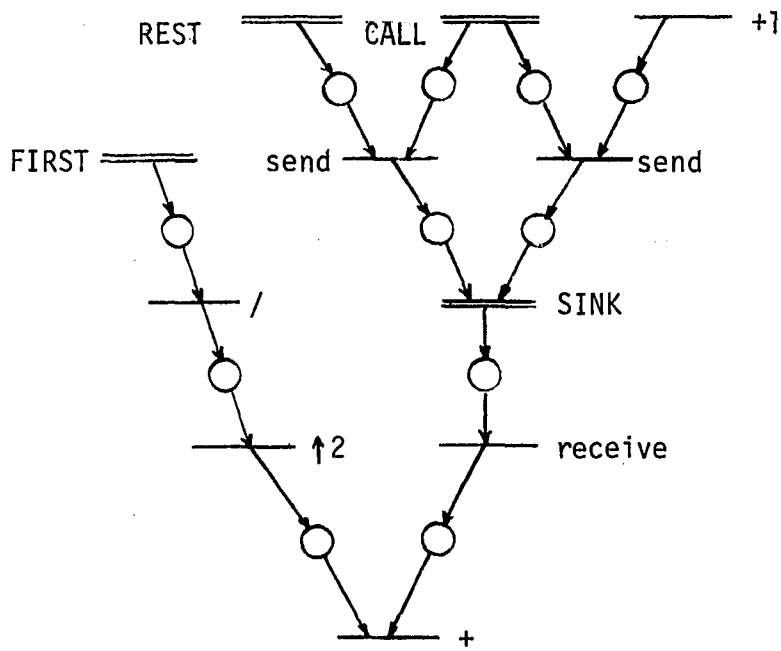
Figure 5.8. Petri net nodes of TRIG



(d) IF node



(e) THEN node



(f) ELSE node

Figure 5.8. Continued

Since $\alpha = 1/(m + 1)$,

$$t_1(SINK) = (m + 1)8 + 1 + m[5 + \max(3,5)]$$

$$= 18(m) + 9$$

Therefore,

$$t_1(STM) = 10 + [18(m) + 9] = 18(m) + 19$$

$$t_1(TRIG) = 3 + [18(m) + 19] = 18(m) + 22$$

Figure 5.9 shows the simulation results for three runs.

RUN	m	time steps
1	2	53
2	3	69
3	5	101

Figure 5.9. Simulation runs for TRIG

Solving using this data yields

$$t_1^{smlate}(TRIG) = 16(m) + 21$$

The dominant discrepancy arises out of the calculation of the delay between the first and second tokens input to the rest operation. The actual value is $d_{1,2} = 3 < p = 5$.

SSUM

This program computes a value for $\sum_{i=1}^{m} [i * \sum_{j=1}^{A_i} j^2]$ for an input stream A. The program consists mainly of a streamed computation with a recursive procedure phase node. The high level code appears in Figure 5.10 and relevant representations of nodes are found in Figure 5.11. The

```
PROCEDURE SSUM
   INTEGER Z,m;
   FILE INFILE,OUTFILE;
   PROCEDURE P (INTEGER STREAM A)
         RETURNS (INTEGER STREAM B)
      B = IF EMPTY(A) THEN EOS
            ELSE BEGIN
                 INTEGER STREAM C,B;
                 INTEGER S;
                 C = SCREATE(1, FIRST(A), 1);
                 S = SUM(C * C)
                 B = CONS[S, P(REST(A))]
                 END (B)
      END

   INPUT m FROM INFILE;
   Z = STREAMED
          INTEGER STREAM A,X,Y;
          INTEGER Z;
          STR-INPUT A FROM INFILE;
          X = SCREATE(1,m,1);
          Y = X * P(A);
          Z = SUM(Y)
          END(Z)
   OUTPUT Z TO OUTFILE
   END
```

Figure 5.10. Program SSUM

following attributes are computed:

$$t_1(SSUM) = t_1(START) + t_1(READ) + t_1(STM) + t_1(WRITE)$$

$$= 2 + 2 + t_1(STM) + 1$$

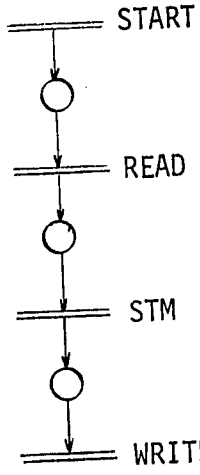
$$= 5 + t_1(STM)$$

$$t_1(STM) = t_1(STR\text{-}INPUT) + t(send) + t_1(P)$$

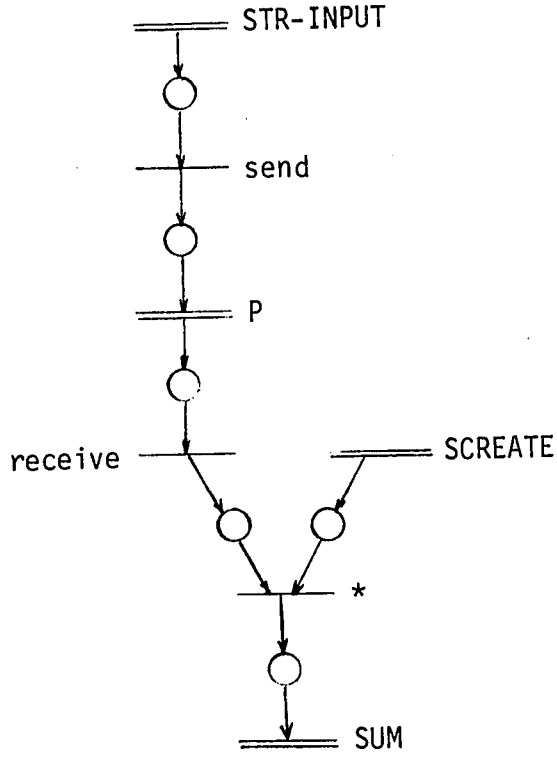
$$+ t(receive) + t(*) + t_1(SUM)$$

$$= 3 + 1 + t_1(P) + 1 + 1 + 2 + m \quad max(3,p)$$

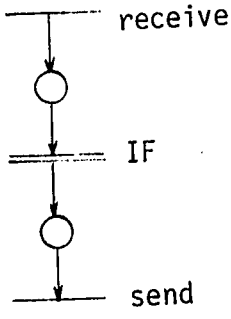
$$= 8 + t_1(P) + m \quad max(3,p)$$



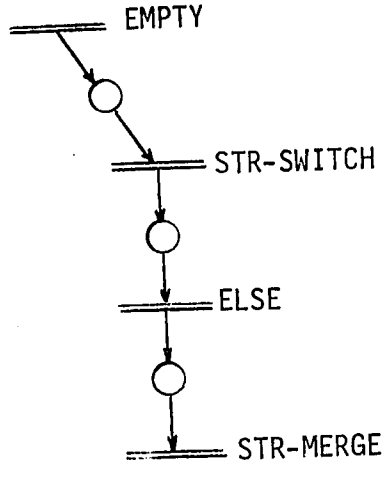
(a) Procedure SSUM



(b) Streamed computation node STM

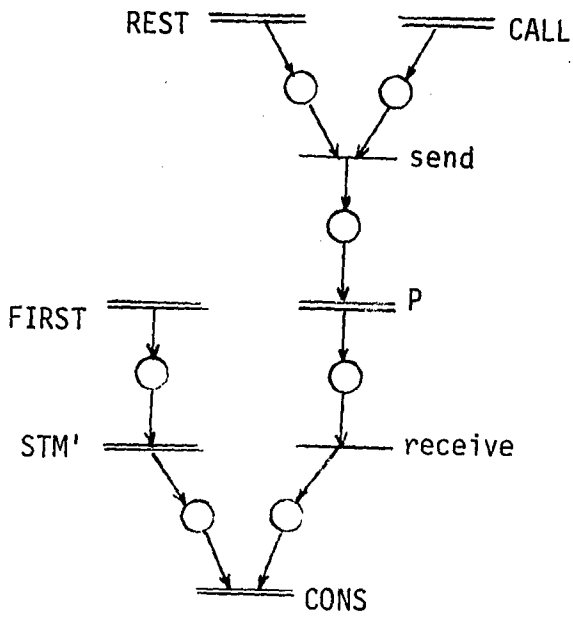


(c) Procedure P

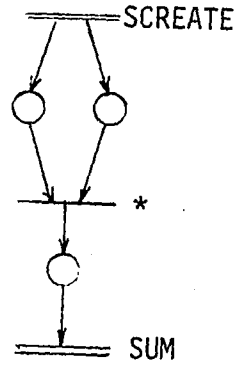


(d) f_{IF} node

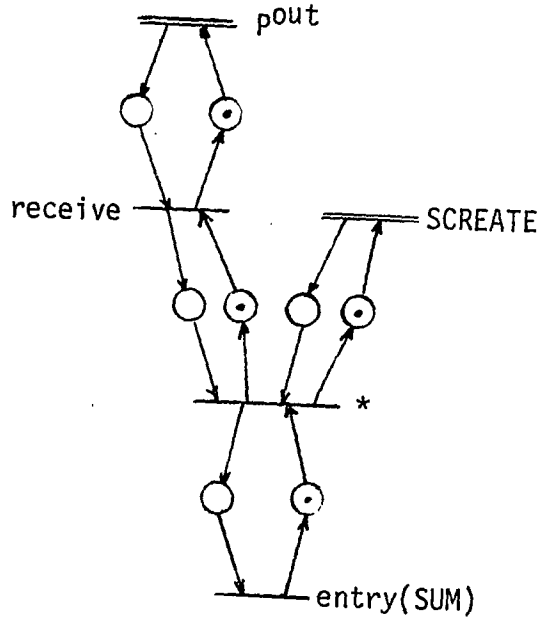
Figure 5.11. Petri net nodes of SSUM



(e) ELSE node

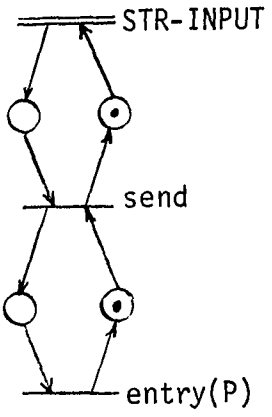


(f) Streamed computation node STM'

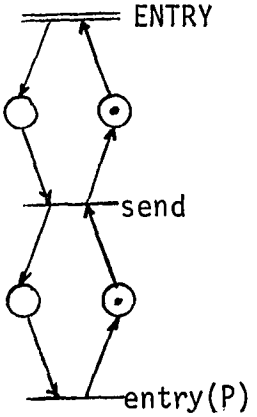


(g) Petri net $STMp^{out}$

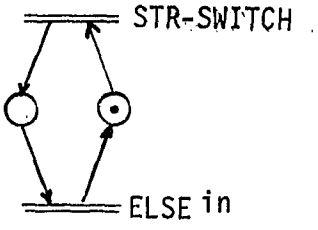
Figure 5.11. Continued



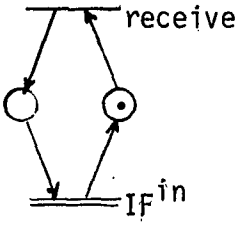
(h) Petri net $STMp^{in}$



(i) Petri net $ELSEp^{in}$



(j) Petri net IFp^{in}



(k) Petri net Pp^{in}

Figure 5.11. Continued

$$t_1(P) = t(receive) + t_1(IF) + t(send)$$

$$= 1 + t_1(IF) + 1$$

$$= 2 + t_1(IF)$$

$$t_1(IF) = \tau(^{f}IFs_1)$$

$$= t_1(EMPTY) + t_1(STR\text{-}SWITCH) + t_1(ELSE) + t_1(STR\text{-}MERGE)$$

$$= 3 + 2 + t_1(ELSE) + 1$$

$$= 6 + t_1(ELSE)$$

$$t_1(ELSE) = t_1(FIRST) + t_1(STM') + t_1(CONS)$$

$$= 2 + t_1(STM') + 1$$

$$= 3 + t_1(STM')$$

$$t_1(STM') = t_1(SCREATE) + t(*) + t_1(SUM)$$

$$= 5 + 1 + 2 + m' \cdot \max(3,p')$$

$$= 8 + m' \cdot \max(3,p')$$

p' is computed as

$$p' = \max[\tau(STM'p), IP(SCREATE)]$$

$$= \max(2,4) = 4$$

Therefore,

$$t_1(STM') = 8 + 4(m')$$

$$t_1(ELSE) = 11 + 4(m')$$

$$t_1(IF) = 17 + 4(m')$$

$$t_1(P) = 19 + 4(m')$$

$$t_1(STM) = 27 + 4(m') + m \cdot \max(3,p)$$

p is computed as

$$p = \max[\tau(STMp^{out}), IP_{out}(P), IP(SCREATE)]$$
$$= \max[2, IP_{out}(P), 4]$$

$IP_{out}(P)$ is dominated by the delay in making a recursive invocation and returning successive stream values and is computed as

$$IP_{out}(P) = [t(receive) + t_1(EMPTY) + t_1(STR\text{-}SWITCH) + t_1(REST) + t(send)] + [t(receive) + t_1(CONS) + t_1(STR\text{-}MERGE) + t(send)]$$
$$= 13 + \max(3,p'')$$

p" is updated to capture the influence of $IP_{in}(P)$ which is computed as

$$IP_{in}(P) = \max[\tau(Pp^{in}), IP(IF^{in})]$$
$$= \max[2, IP(IF^{in})]$$
$$IP(IF^{in}) = \max[\tau(^{f}IFp^{in}), IP(STR\text{-}SWITCH), IP(ELSE^{in})]$$
$$= \max[2, 3, IP(ELSE^{in})]$$
$$IP(ELSE^{in}) = \max[\tau(ELSEp^{in}), IP(ENTRY)]$$
$$= \max[2, IP(ENTRY)]$$

where ENTRY is the coalesced nodes FIRST and REST and

$$IP(ENTRY) = \max[IP(FIRST), IP(REST)]$$
$$= \max(3,3) = 3$$

Therefore,

$$IP_{in}(P) = 3$$

and p" is replaced by $\max[p''', IP_{in}(P)]$, where p''' is the period of the stream input to the procedure P.

Substituting,

$$p = IP_{out}(P) = 13 + \max(3,p''')$$

$$t_1(STM) = 27 + 4(m') + m \quad \max[3,\ 13 + \max(3,p\ )]$$

p''' is computed as

$$p''' = \max[\ \tau(STM p^{in}),\ IP(STR\text{-}INPUT)]$$

$$= \max[2,5] = 5$$

Therefore,

$$t_1(STM) = 27 + 4(m') + m \quad [13 + \max(3,5)]$$

$$= 27 + 4(m') + 18(m)$$

$$t_1(SSUM) = 32 + 4(m') + 18(m)$$

where m is the length of the input stream and m' is computed as the expected value of a token of the input stream. Figure 5.12 shows the simulation results for four runs.

RUN	m	m'	time steps
1	2	2	58
2	3	2	74
3	5	2	1Ø6
4	2	3	62

Figure 5.12. Simulation results for SSUM

Note: for these runs the value for m' was fixed by inputting a stream where all tokens had a value of m'. These actual runs yield a timing equation of

$$t_1^{smlate}(SSUM) = 18 + 4(m') + 16(m)$$

The discrepancy in the coefficient of the m term is due to the assumption that $d_{1,2}$ = p" in the period of the stream input to the rest node. The scalar discrepancy is due to the assumption that $d_{last,eos}$ for both streamed computations is equal to their respective periods.

MULT

This program computes a value for a convolution and consists primarily of two streamed computations, the first producing values (buffered streams) for the second. Each contains a recursive streamed procedure that on each invocation processes the entire input streams to produce a single token of the output stream. The high level code appears in Figure 5.13. Pertinent graphical representations appear in Figure 5.14. Attribute values are computed as follows:

$$
\begin{aligned}
t_1(\text{MULT}) &= t_1(\text{START}) + t_1(\text{READ}) + t_1(\text{STM1}) + t_1(\text{STM2}) + \\
&\quad t_1(\text{WRITE}) \\
&= 2 + 2 + t_1(\text{STM1}) + t_1(\text{STM2}) + 1 \\
&= 5 + t_1(\text{STM1}) + t_1(\text{STM2}) \\
t_1(\text{STM1}) &= \max[t_1(\text{STR-INPUT}),\ t_1(\text{CALL}), \\
&\qquad t_1(\text{SCREATE}) + t(\text{identity})] \\
&\quad + t(\text{send}) + t_1(\text{P}) + t(\text{receive}) + t_1(\text{BUF})] \\
&= \max[3,\ 4,\ 5 + 1] + 1 + t_1(\text{P}) + 1 + \\
&\quad [3 + \text{npl} \cdot \max(4,p)] \\
&= 11 + t_1(\text{P}) + \text{npl} \cdot \max(4,p)
\end{aligned}
$$

where p is the period of the stream input to the BUF operation.

$$t_1(P) = t(\text{receive}) + t_1(IF) + t(\text{send})$$

$$= 1 + t_1(IF) + 1 = 2 + t_1(IF)$$

$$t_1(IF) = \tau(^{f}IFs_1)$$

$$= t(>) + t_1(\text{STR-SWITCH}) + t_1(\text{ELSE}) + t_1(\text{STR-MERGE})$$

$$= 1 + 2 + t_1(\text{ELSE}) + 1$$

$$= 4 + t_1(\text{ELSE})$$

```
PROCEDURE MULT
   FILE INFILE,OUTFILE;
   REAL ARRAY AXX,AX2,AX3;
   REAL Z;
   INTEGER npl
   INPUT npl FROM INFILE;
   AXX,AX2,AX3 = STREAMED
      REAL ARRAY AXX,AX2,AX3;
      REAL STREAM XX,X2,X3;
      PROCEDURE P (INTEGER K, NPL, REAL STREAM XX, XZ)
            RETURNS (REAL STREAM X3)
         X3 = IF K > NPL THEN EOS
                 ELSE BEGIN
                    INTEGER M;
                    REAL X;
                    REAL STREAM Y;
                    M = NPL - K + 1;
                    X = SUM[SUBSTM(XX, 0, M) *
                            SUBSTM(X2, K - 1, M)] / 2;
                    Y = CONS[X, P(K + 1, NPL, XX, X2)]
                    END (Y)
         END

      STR-INPUT XX FROM INFILE;
      X2 = FLOAT[SCREATE(1, npl, 1)];
      X3 = P(1,npl,XX,XZ);
      AXX = BUF(XX);
      AX2 = BUF(X2);
      AX3 = BUF(X3);
      END (AXX,AX2,AX3)
```

Figure 5.13. Program MULT

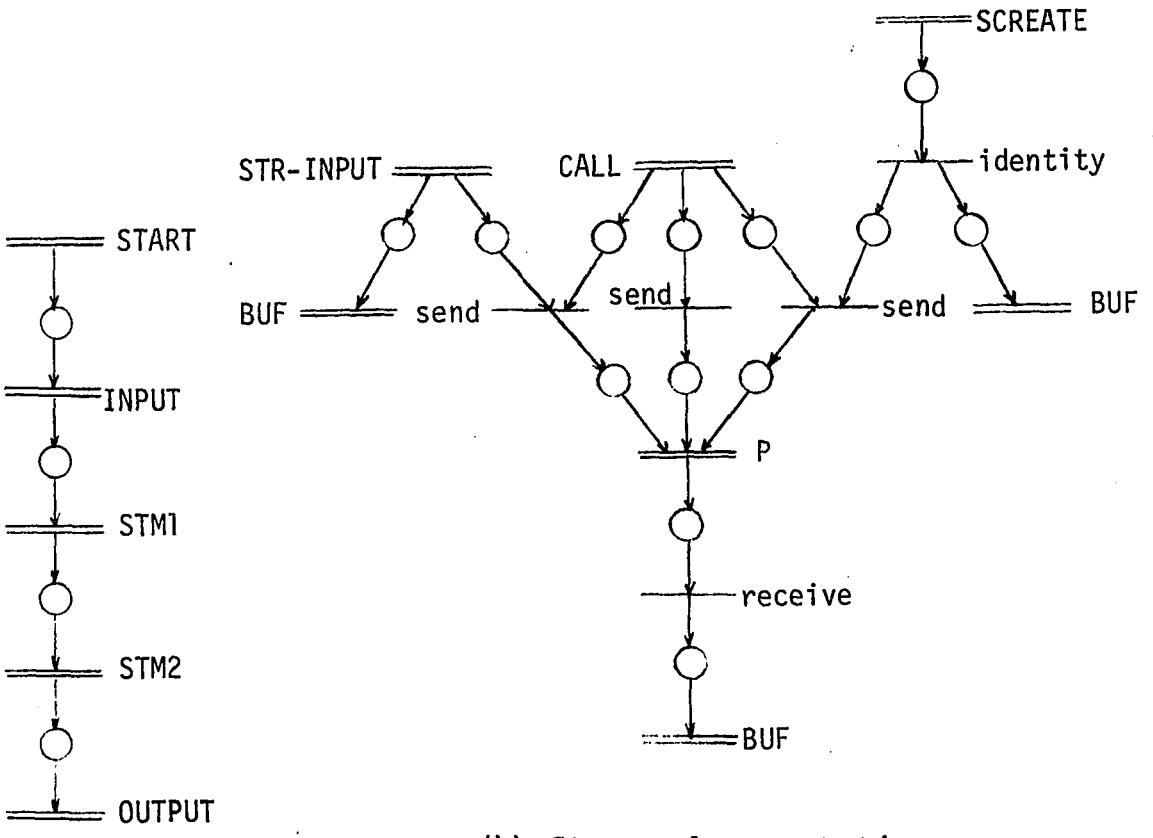
```
Z = STREAMED
        REAL STREAM XX,X2,X3,R;
        INTEGER W;
        PROCEDURE Q (INTEGER K, NPL, REAL STREAM XX,X2)
              RETURNS (REAL STREAM R)
           R = IF K > NPL THEN EOS
                 ELSE BEGIN
                    REAL X;
                    REAL STREAM Y;
                    X = SUM[SUBSTM(XX, Ø, K - 1) *
                          SUBSTM(X2, Ø, K - 1)] / 2;
                    Y = CONS[X, Q(K + 1, NPL, XX, X2)]
                    END (Y)
           END

        XX = UNBUF(AXX);
        X2 = UNBUF(AX2);
        X3 = UNBUF(AX3);
        R = Q(1, npl, XX, X2);
        W = SUM(X3 + R)
        END (W)

OUTPUT Z TO OUTFILE

END
```

Figure 5.13. Continued

$$t_1(\text{ELSE}) = t(-) + t(+) + t_1(\text{SUBSTM2}) + t(*) + t_1(\text{SUM}) + t(/) + t_1(\text{CONS})$$

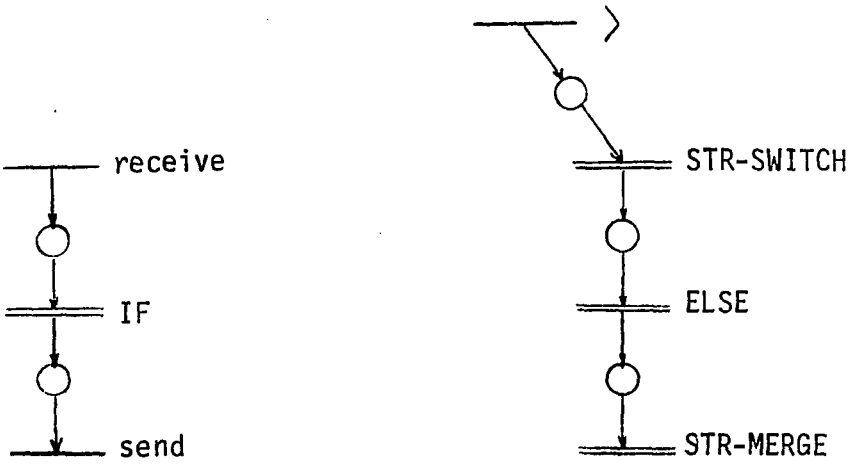
$$= 1 + 1 + [7 + k \cdot \max(3,p')] + 1 + [2 + m \cdot \max(3,p')] + 1 + 1$$

$$= 14 + (k - 1) \cdot \max(3,p') + m \cdot \max(3,p')$$



(a) Procedure MULT

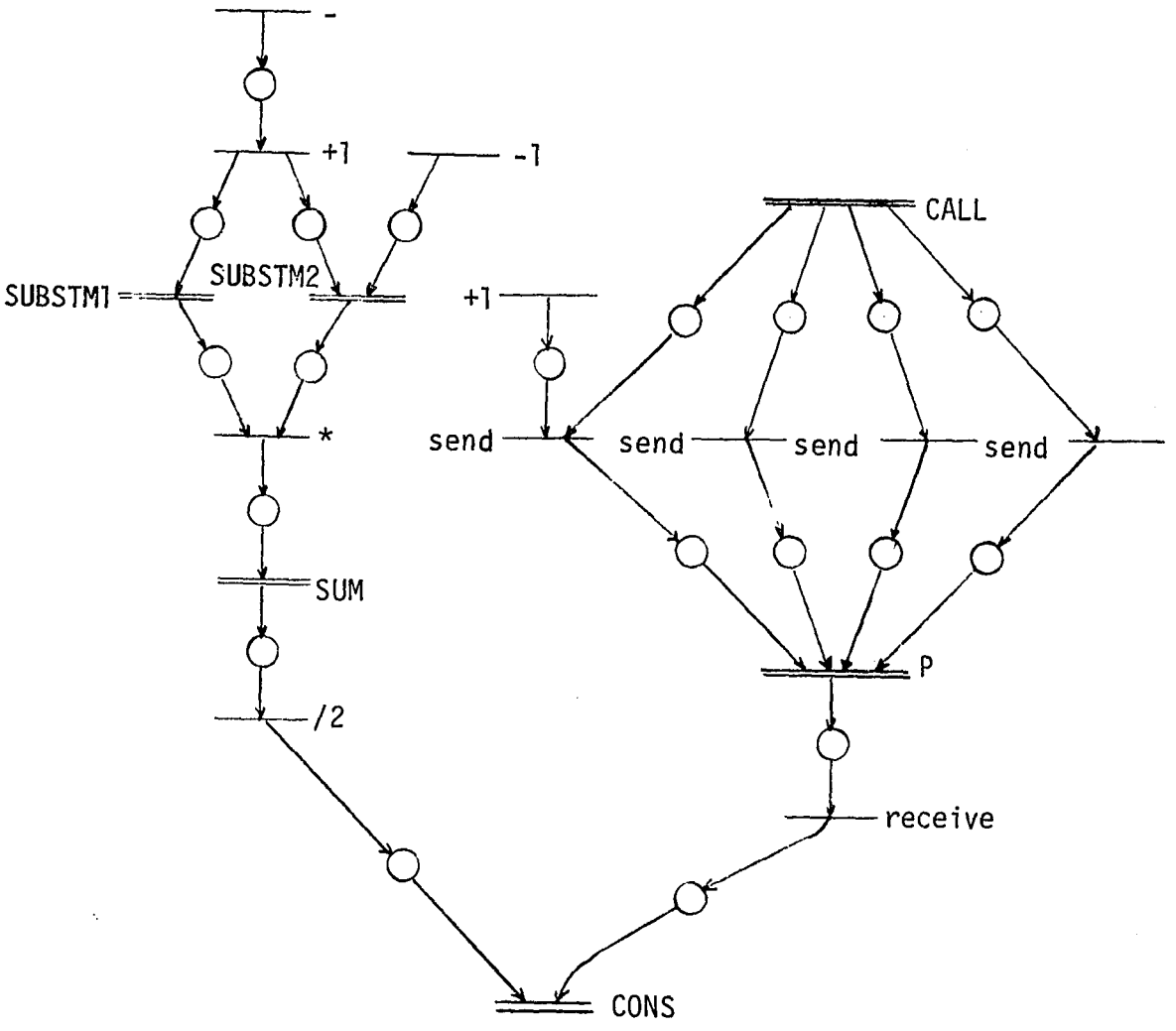
(b) Streamed computation node STM1



(c) Procedure P

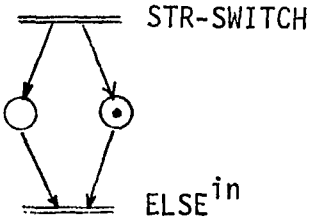
(d) ^{f}IF node

Figure 5.14. Petri net nodes of MULT

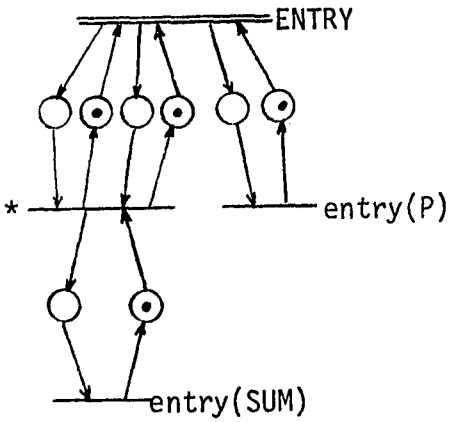


(e) ELSE node

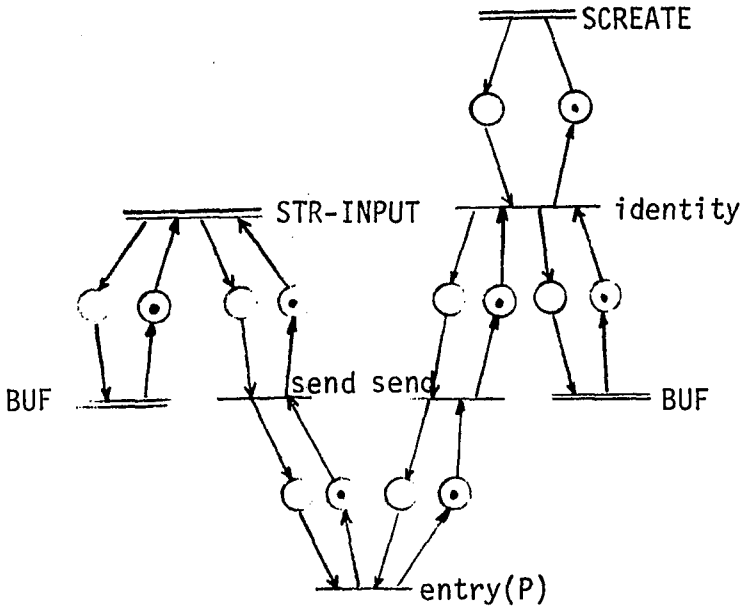
Figure 5.14. Continued



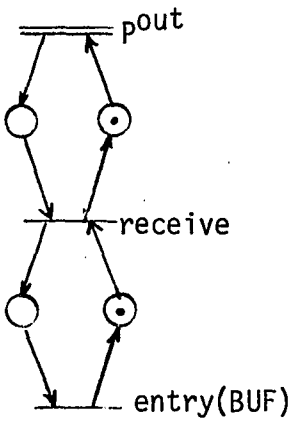
(f) Petri net ^{f}IFp



(g) Petri net ELSEp



(h) Petri net STM1p



(i) Petri net STM1p

Figure 5.14. Continued

Since m = npl - k + 1,

$$t_1(\text{ELSE}) = 14 + npl \cdot \max(3,p')$$

where p' is the period of the stream input to the sum operation. p' is updated to capture the influence of $IP_{in}(P)$ which is computed as:

$$IP_{in}(P) = \max[\tau(Pp^{in}), IP(IF^{in})]$$
$$= \max[2, IP(IF^{in})]$$
$$IP(IF^{in}) = \max[\tau(IFp^{in}), IP(\text{STR-SWITCH}), IP(ELSE^{in})]$$
$$= \max[2, 3, IP(ELSE^{in})]$$
$$IP(ELSE^{in}) = \max[\tau(ELSEp^{in}), IP(\text{ENTRY})]$$
$$= \max[2, IP(\text{ENTRY})]$$

where ENTRY is the coalesced nodes of SUBSTM1, SUBSTM2, send (for XX), and send (for X2), and

$$IP(\text{ENTRY}) = \max[IP(\text{SUBSTM1}), IP(\text{SUBSTM2})]$$
$$= 3$$

Therefore,

$$IP_{in}(P) = 3$$

and p' is replaced by $\max[p'', IP_{in}(P)]$, where p" is the period of the stream input to the procedure P.

Substituting,

$$t_1(\text{ELSE}) = 14 + npl \cdot \max(3,p'')$$

$$t_1(P) = 20 + npl \cdot \max(3,p'')$$

p" is computed as

$$p'' = \max[\tau(STM1p^{in}), IP(\text{STR-INPUT}), IP(\text{BUF}), IP(\text{SCREATE})]$$
$$= \max(2,5,4,4) = 5$$

Consequently,

$$t_1(P) = 20 + 5(npl)$$

and

$$t_1(STM1) = 11 + [20 + 5(npl)] + npl \cdot \max(4,p)$$

p again is the period of the stream input to the BUF operation and is computed as

$$p = \max[\tau(STM1p^{out}), IP_{out}(P)]$$
$$= \max[2, IP_{out}(P)]$$

$IP_{out}(P)$ is dominated by the delay in making a recursive invocation and returning successive stream values and is computed as

$$IP_{out}(P) = [t(receive) + t(>) + t_1(STR\text{-}SWITCH) + t_1(CALL) + t(send)] + [t(receive) + t_1(CONS) + t_1(STR\text{-}MERGE) + t(send)]$$
$$= 13$$

Therefore, p = 13 and

$$t_1(STM1) = 31 + 18(npl)$$

$t_1(STM2)$ is computed in a similar manner as

$$t_1(STM2) = 29 + 18(npl)$$

and substituting these values yields

$$t_1(MULT) = 5 + [31 + 18(npl)] + [29 + 18(npl)]$$
$$= 65 + 36(npl)$$

Figure 5.15 shows the simulation results for four runs.

RUN	npl	time steps
1	2	66
2	3	84
3	4	1Ø3
4	5	121

Figure 5.15. Simulation results for MULT

Solving using this data yields approximately

$$t_1^{smlate}(MULT) = 3Ø + 18(npl)$$

The major reason the approximated value for $t_1(MULT)$ differs so much from the actual value is a result of Constraint 2 introducing a high degree of error; i.e., there is a significant amount of overlap between the two streamed computations. In fact, to a large extent the stream R is readily available in the streamed computation STM2 by the time the buffered stream AX3 arrives due to the early availability of AXX and AX2. Furthermore, it is believed that larger values of npl (not simulated due to economic factors) would increase the coefficient of npl in $t_1^{smlate}(MULT)$ possibly as high as 22. This is due to not saturating the pipelined computations.

Summary

Figure 5.16 summarizes the five simulations and their analysis. A percent error is calculated on dominant terms only.

The method of top down analysis for the feedback architecture via recursive nodal reductions appears to be a fairly good approach to the analysis of programs and their behavior with the exception of the cases where streamed computations are not clearly defined and do not satisfy Constraint 2. Large error factors appear with this type of streamed computation and more research is needed on streams for this type of architecture.

program	$t_1^{computed}$	t_1^{smlate}	% error
HYSL	19+4m+ n(63+16m)	19+4m+ n(47+16m)	Ø
NEXP	67n+8	66n+8	+1.5
TRIG	18m+22	16m+21	+12.5
SSUM	32+4m'+18m	18+4m'+16m	+12.5
MULT	65+36m	3Ø+18m	+1ØØ.Ø

Figure 5.16. Summarization of simulations

CHAPTER VI. OTHER ARCHITECTURES

The basic techniques described in previous chapters are applicable to the analysis of other parallel architectures, with, of course, appropriate modifications. Generally, abstract operations are represented as transitions and arcs representing data and/or control signal dependencies connect these transitions.

The recursive stream-oriented model is based on the architecture proposed by Dennis and Weng [Dennis and Weng 1979]. The language for this machine is restricted to a degree that "feedback" signals are not necessary, thereby eliminating a high degree of overhead. These restrictions insure that no static copy of any instruction is ever enabled more than once. All iterative computations (e.g., the while-do) are prohibited and the expression of these computations are made recursively (for this research this is assumed to be done via recursive procedures. Other expressions could be made via specialized constructs; e.g., the "for-iter" construct [Ackerman and Dennis 1978]). Additionally, streams are built up as linearized structures (see Figure 6.1). The operations FIRST and REST are simply implemented as "select" operations on the structure. In order to allow concurrent operations on successive stream tokens, the concept of "holes" is implemented [Dennis and

Weng 1979]. When a CONS operation is performed (see Figure 6.2) the "scalar" value v is appended to a structure and a "hole" is created for the remainder of the stream, T. The stream S, created by the CONS operation, is immediately available while operations on the "rest of S" are suspended until the hole has been filled by a "write-hole" operation (the signal is used in the management of procedure activations).

Non-streamed constructs in the stream-oriented model are analyzed using the methodology introduced for the feedback model (the while-do is eliminated). However, the general approach is simplified for the streamed computa-

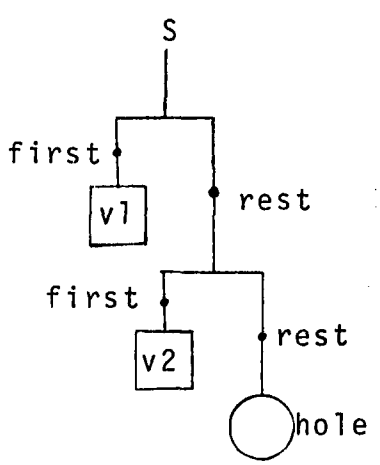
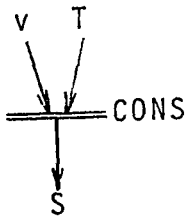


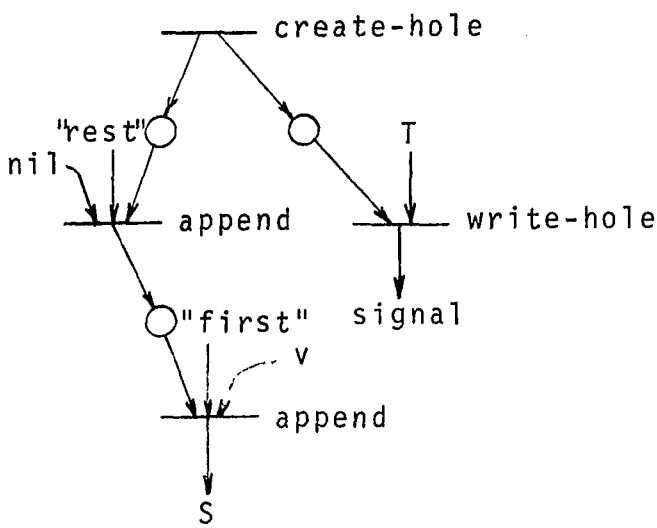
Figure 6.1. Stream as a linear structure

tions. No Np-type Petri nets need be constructed since feedback signals are not present. This eliminates the need for t_2, n, entry, and exit attributes. Since the only operations defined on streams (other than preanalyzed constructs) are either treated as scalar operations (e.g., FIRST and REST) or are recursive streamed procedures, only t_1 and IP attributes are required.

Before discussing the analysis of streamed computations, consider the timing behavior of the recursive



(a) Abstract operation



(b) Implementation of CONS

Figure 6.2. CONS operation in recursive stream-oriented model

streamed procedure of Figure 6.3 (compare with Figure 4.37). A timing diagram appears in Figure 6.4. As indicated in this diagram, the creation of holes precludes the necessity of transmitting every stream token through every previous invocation (this is perhaps the most significant aspect of this architecture). The period of availability of successive tokens of the output stream, IP(P), is determined by the abstract value $t_1(R) / a$ where R represents the computation required to construct arguments for, and make the recursive invocation of, a recursive call and a represents the number of tokens created by each invocation. The dynamic code unraveling that occurs as successive invocations of P are made is illustrated in Figure 6.5 (the CONS operations are not reduced). Using the method previously described for the feedback model, the calculation of $t_1(P)$ that involves a CONS operation proceeds in a

```
PROCEDURE P (INTEGER STREAM S)
      RETURNS (INTEGER STREAM T)
   T = IF EMPTY(S) THEN EOS
         ELSE BEGIN
              INTEGER STREAM R;
              R = CONS{f[FIRST(S)], P[REST(S)]}
              END (R)
   END
```

Figure 6.3. Sample streamed recursive procedure

straight-forward manner as

$$t_1(P) = t(\text{receive}) + t_1(\text{EMPTY}) + t(\text{switch}) + t_1(\text{FIRST}) + t_1(f) + t_1(\text{CONS}) + t(\text{send})$$

Assuming $t_1(\text{EMPTY}) = 2$ and the FIRST and REST operations are implemented as simple "select" operations,

$$t_1(P) = 1 + 2 + 1 + 1 + t_1(f) + 1 + 1$$
$$= 7 + t_1(f)$$

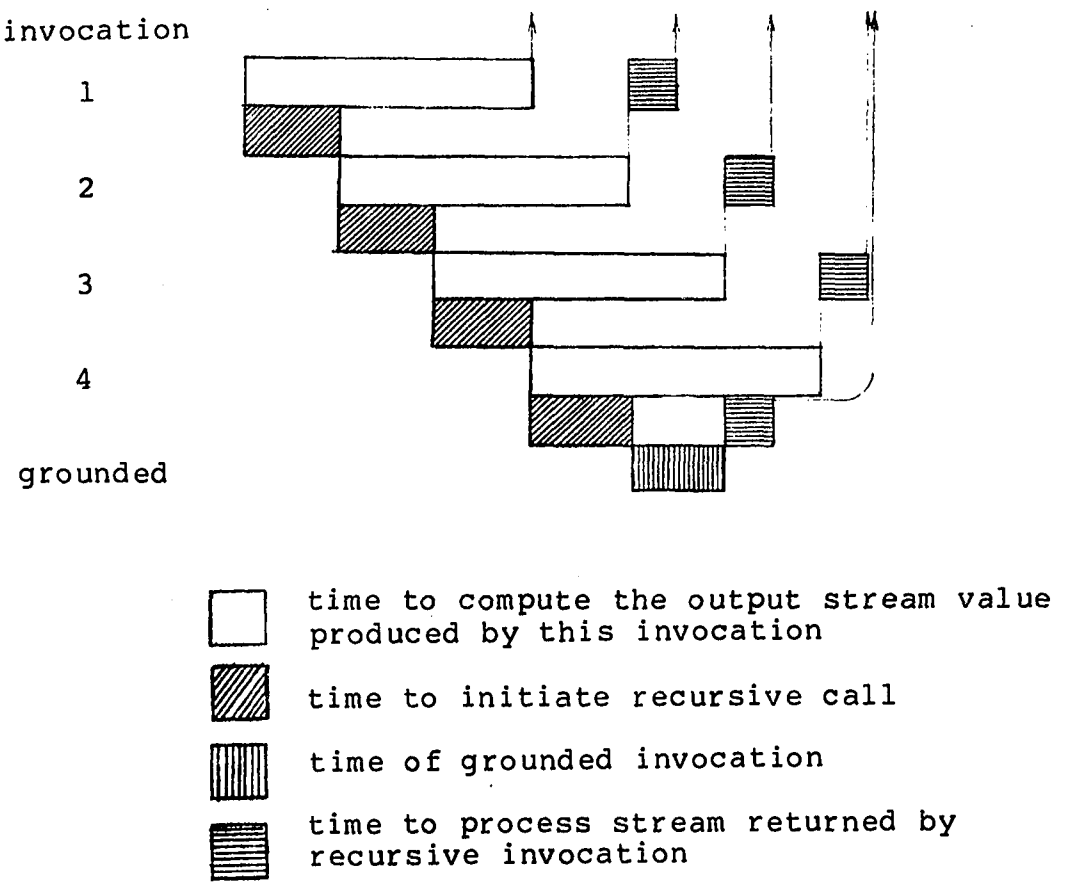


Figure 6.4. Timing diagram of streamed recursive procedure

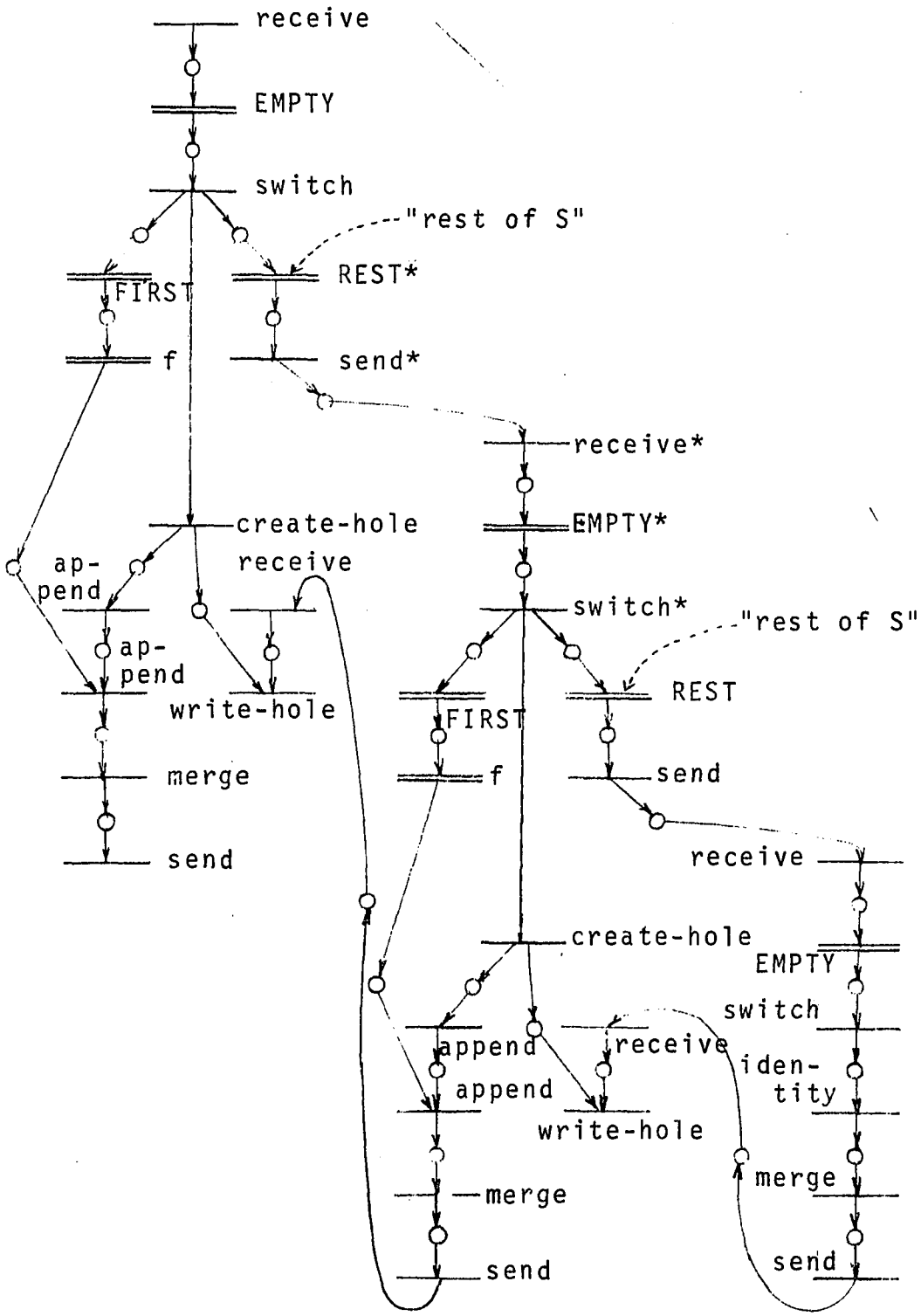


Figure 6.5. Dynamic code unraveling of streamed recursive procedure (starred operations used in calculation of delay of firing REST node)

The determination of the value IP(P) is not accomplished as easily. Referring to Figure 6.5, this value is seen to be

$$\begin{aligned} IP(P) &= [t(receive) + t_1(EMPTY) + t(switch) + \\ &\quad t_1(REST) + t(send)] \;/\; 1 \\ &= 1 + 2 + 1 + (1 + d) + 1 \\ &= 6 + d \end{aligned}$$

where $t_1(REST) = 1 + d$ and d is the delay incurred in waiting on the write-hole operation that produces the "rest of S" which is input to this invocation. This value in most situations is substantially less than the period p between successive tokens of the stream input to the procedure P. This is conceptually understood by noting that a certain amount of computation has been done between the time the "first" value was released and the time the REST operation begins waiting. For the current example, the d value for the REST operation of the second invocation (or any subsequent invocation) is computed as

$$\begin{aligned} d = \max\{0,\ p - [&t(select) + t(send) + \\ &t(receive) + t_1(EMPTY) + t(switch)]\} \end{aligned}$$

The value t(select) is the nodal firing time of the REST operation of the previous invocation once the appropriate hole has been filled. The filling of the hole of the input stream allows the firing of the starred operations in Figure 6.5. Obviously, under certain circumstances the d value may

be substantially less than p (or even Ø). This points out one of the main advantages of this architecture in terms of execution time performance over the feedback model.

Should the procedure P be found in a streamed computation as appears in Figure 6.6 where IP(source) = 7, the period of the stream input to the sink node, IP(P), is computed as

$$\begin{aligned} IP(P) &= 6 + d \\ &= 6 + \max(\emptyset, p - 6) \\ &= 6 + \max(\emptyset, 7 - 6) \\ &= 7 \end{aligned}$$

The calculation of the appropriate d values is complicated by the necessity to analyze portions of several nodes

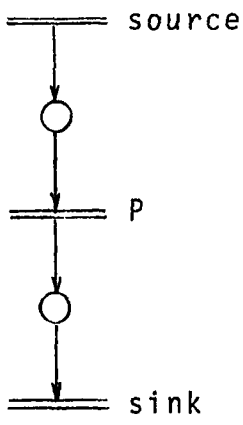


Figure 6.6. Streamed computation

as a single entity. In this example, it involves portions of two invocations, the else body of the conditional of the first invocation, and a portion of the conditional of the second invocation. The approximation of d = p could be made, but this is expected to drastically inflate the aproximated execution time of streamed computations. A sophisticated analysis that allows the appropriate calculation of IP(P) is in order and may be more easily accomplished by analyzing sample unraveled portions of recursive procedure code as appears in Figure 6.5. However, for the purposes of this research, the value of d will be left as a parameter to the analysis.

As a final example, consider the program N found in Figure 6.7. This program computes $\sum_{j=1}^{m} X^j/i$ for input values X and m using a streamed computation. The nodal reductions appear in Figure 6.8. As previously noted, "scalar" computations are analyzed through the use of Ns_1 Petri nets resulting in

$$t_1(N) = t(input) + t(input) + t_1(STM) + t(output)$$
$$= 1 + 1 + t_1(STM) + 1$$
$$= 3 + t_1(STM)$$

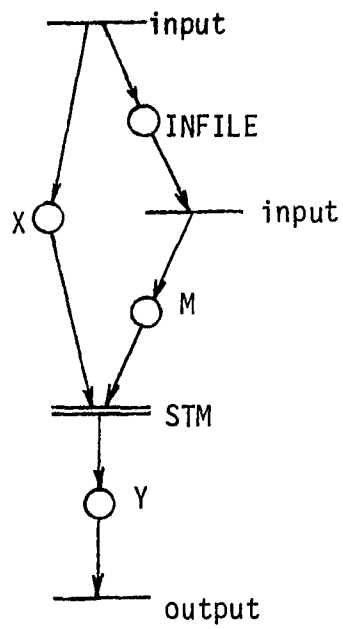
Since all streamed computations are composed of recursive procedure nodes (and/or pre-analyzed streamed operations implemented recursively), the streamed computation is partitioned into many (three in this example) components

```
PROCEDURE P (INTEGER I,M)
     RETURNS (INTEGER STREAM S)
   S = IF I > M THEN EOS
          ELSE BEGIN
               INTEGER STREAM V
               V = CONS[I, P(I+1, M)]
               END(V)
   END

PROCEDURE Q (INTEGER STREAM S, REAL X)
     RETURNS (REAL STREAM T)
   T = IF EMPTY(S) THEN EOS
          ELSE BEGIN
               REAL STREAM U
               INTEGER I
               I = FIRST(S)
               U = CONS{X ↑ I / I, Q[REST(S), X]}
               END(U)
   END

PROCEDURE R (REAL STREAM T)
      RETURNS (REAL Y)
   Y = IF EMPTY(T) THEN Ø.Ø
          ELSE BEGIN
               REAL Z
               Z = FIRST(T) + R[REST(T)]
               END(Z)
   END

INTEGER M
REAL X,Y
INPUT X,M FROM INFILE
Y = STREAMED
    INTEGER STREAM S
    REAL STREAM T
    REAL Z
    S = P(1, M)
    T = Q(S, X)
    Z = R(T)
    END(Z)
OUTPUT Y TO OUTFILE
END
```

Figure 6.7. Sample program N

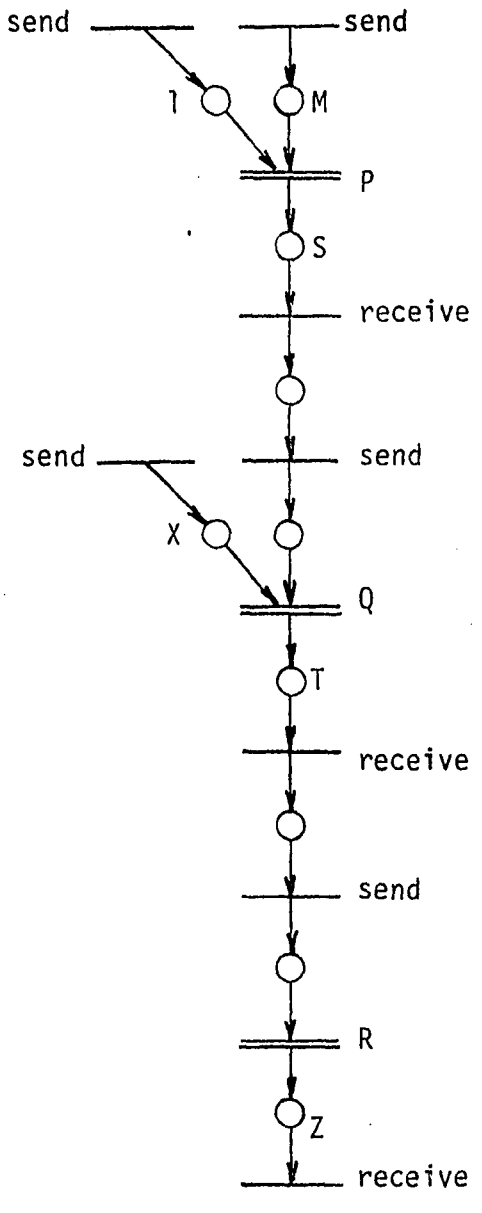
with loose interconnections thereby eliminating the need for Np-type Petri nets. The period p of the stream input to Q is computed as IP(P) and the period p' of the stream input to R is computed as IP(Q). All three procedure nodes are recursive procedures where the then bodies of the enclosed conditionals are encountered only on the grounded invocations. The unraveling of P is shown in Figure 6.8(c) and attributes are computed as

$$t_1(P) = t(\text{receive}) + t(>) + t(\text{switch}) + t(\text{create-hole}) + t(\text{append}) + t(\text{append}) + t(\text{merge}) + t(\text{send}) = 8$$



(a) Program node N

Figure 6.8. Nodal reductions



(b) Streamed computation STM

Figure 6.8. Continued

(c) Unraveling of P

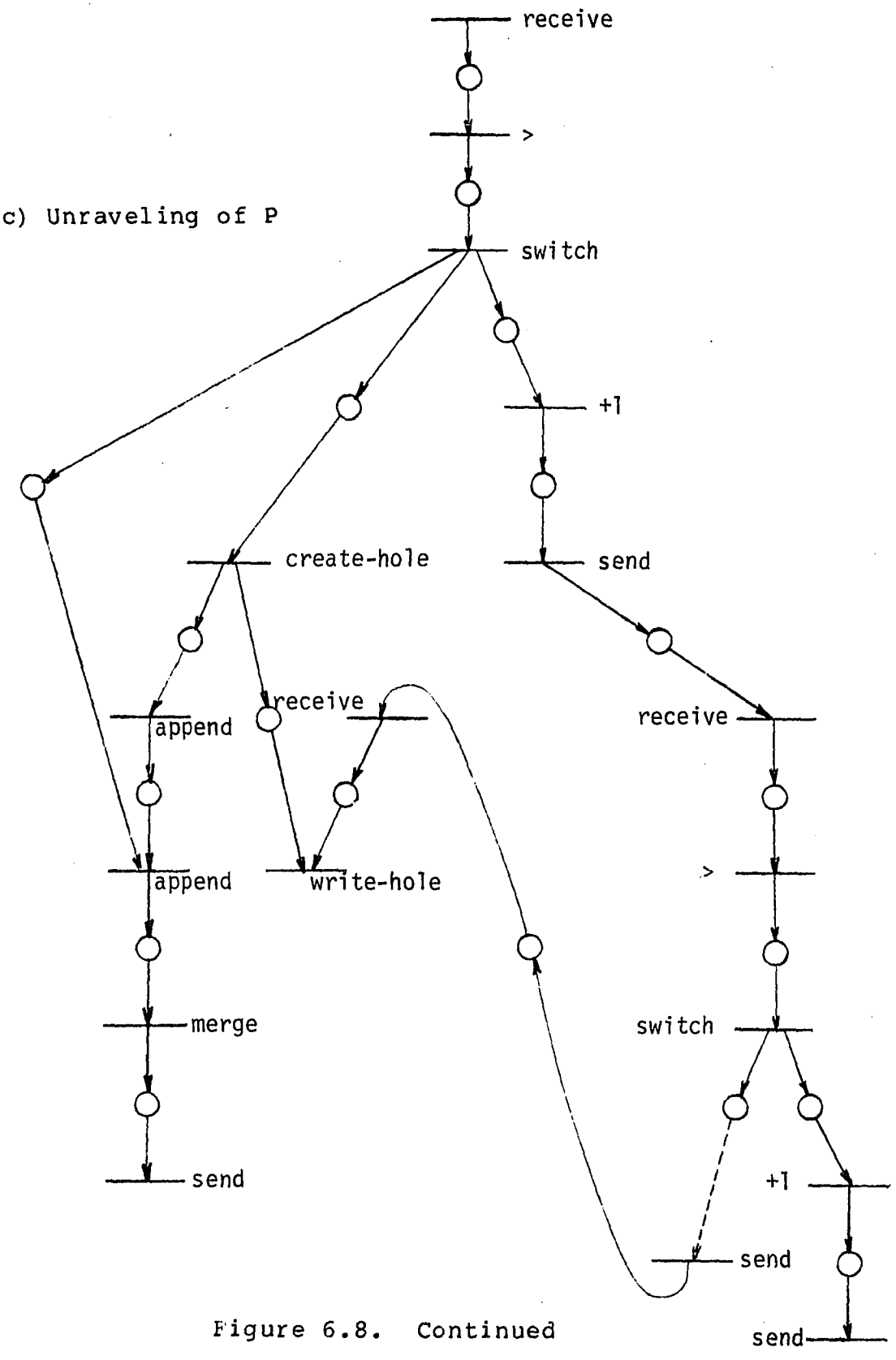


Figure 6.8. Continued

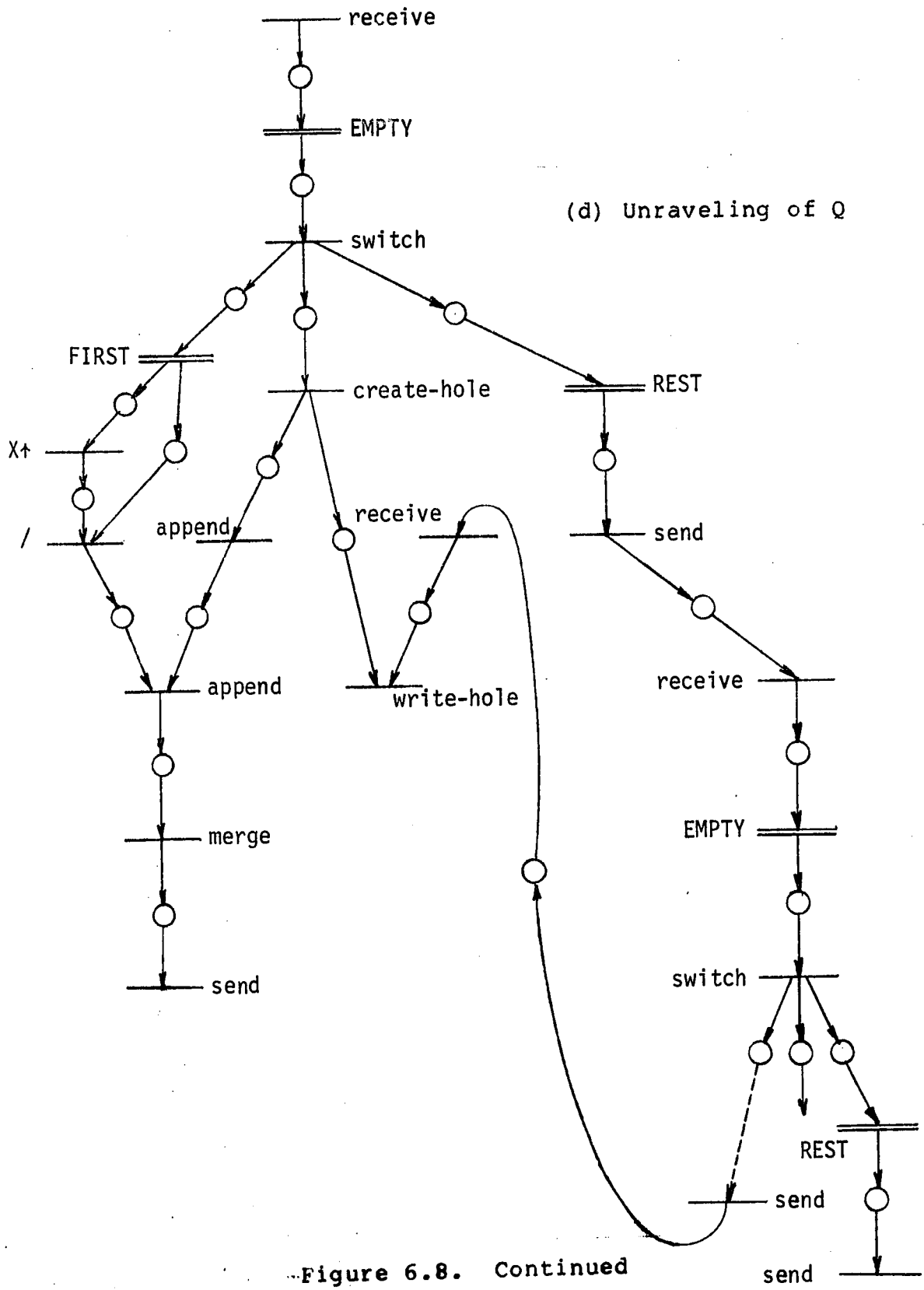
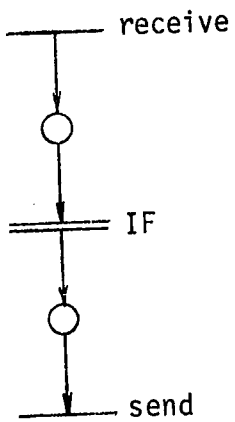
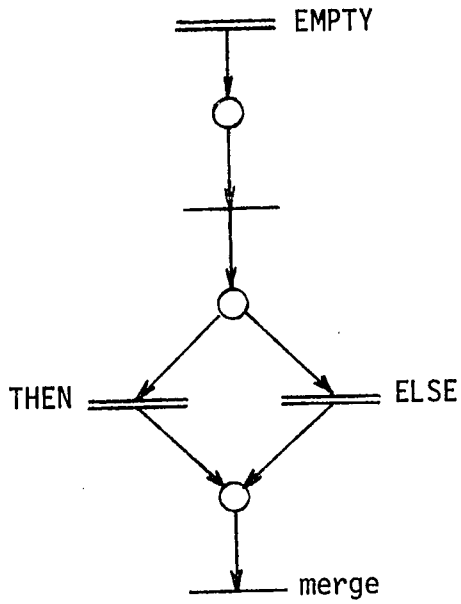


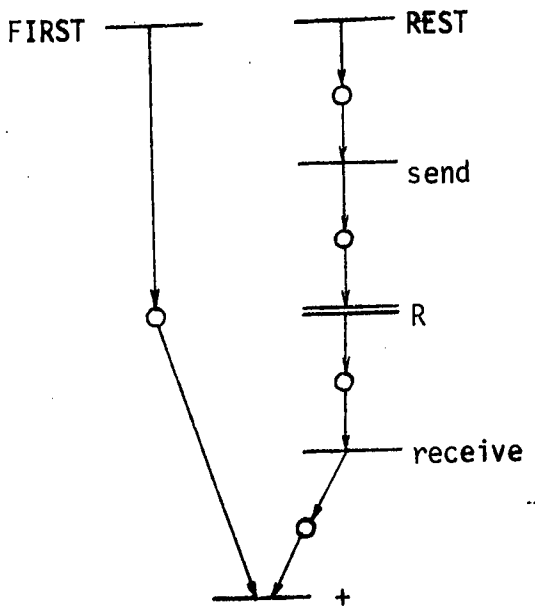
Figure 6.8. Continued



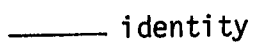
(e) Procedure R



(f) IF node

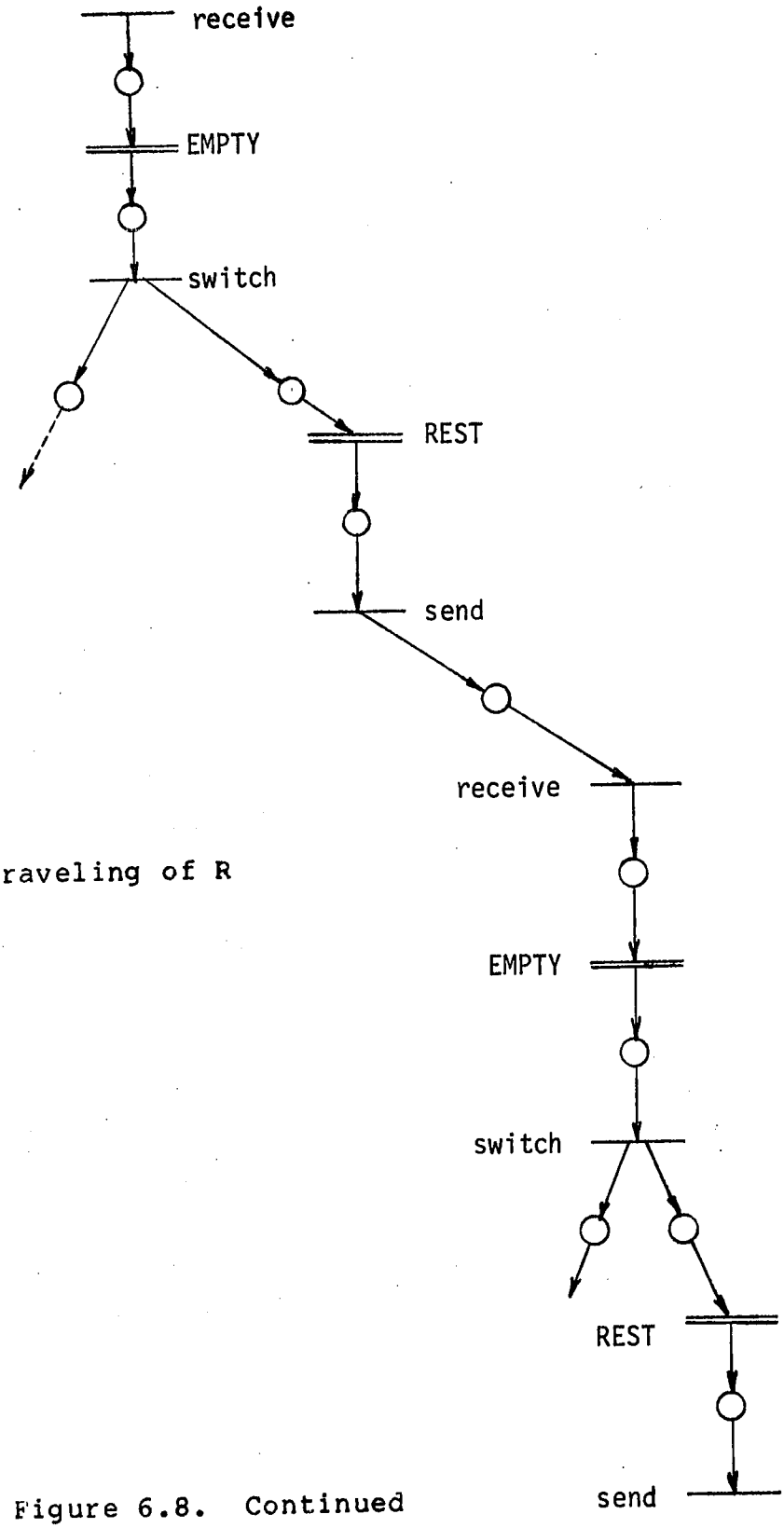


(g) ELSE node



(h) THEN node

Figure 6.8. Continued



(i) Unraveling of R

Figure 6.8. Continued

(assuming unit execution time for all base level operations).

$$IP(P) = t(receive) + t(>) + t(switch) + t(+) + t(send)$$
$$= 5$$

which is the time to make a recursive invocation.

Q is unraveled in Figure 6.8(d) and attributes are computed as

$$t_1(Q) = t(receive) + t_1(EMPTY) + t(switch) + t_1(FIRST) + t(\uparrow) + t(/) + t(append) + t(merge) + t(send)$$
$$= 10$$

assuming $t_1(EMPTY) = 2$.

$$IP(Q) = t(receive) + t_1(EMPTY) + t(switch) + t_1(REST) + t(send)$$
$$= 1 + 2 + 1 + (1 + d) + 1$$
$$= 6 + d$$

where $d = \max(0, p - 6)$ and $p = IP(P) = 5$. As a result, $d = 0$ and

$$IP(Q) = 6.$$

The portion of the streamed computation internal to R is reduced to allow the reduction of R and nodes internal to R as

$$t_1(R) = t(receive) + t_1(IF) + t(send)$$
$$= 1 + t_1(IF) + 1$$
$$= 2 + t_1(IF)$$

$$t_1(IF) = t_1(EMPTY) + t(switch) + \alpha \cdot t_1(THEN) + (1 - \alpha) \cdot t_1(ELSE) + t(merge)$$
$$= 2 + 1 + \alpha \cdot t_1(THEN) + (1 - \alpha) \cdot t_1(ELSE) + 1$$
$$= 4 + \alpha \cdot t_1(THEN) + (1 - \alpha) \cdot t_1(ELSE)$$

$$t_1(THEN) = t(identity) = 1$$

$$t_1(ELSE) = \max[t_1(FIRST) + t(+), t_1(REST) + t(send) + t_1(R) + t(receive) + t(+)]$$
$$= \max[1 + 1, (1 + d) + 1 + t_1(R) + 1 + 1]$$
$$= \max[2, 4 + d + t_1(R)]$$
$$= 4 + d + t_1(R)$$

d is computed from the unraveling of R (Figure 6.8(i)) as

$$d = \max(\emptyset, p - 6)$$

where $p = IP(Q) = 6$. Therefore $d = \emptyset$ and

$$t_1(ELSE) = 4 + \emptyset + t_1(R)$$
$$= 4 + t_1(R)$$

$$t_1(IF) = 4 + \alpha(1) + (1 - \alpha)\ [4 + t_1(R)]$$
$$= 8 - 3\alpha + (1 - \alpha) \cdot t_1(R)$$

$$t_1(R) = 2 + [8 - 3\alpha + (1 - \alpha) \cdot t_1(R)]$$
$$= 10 - 3\alpha + (1 - \alpha) \cdot t_1(R)$$
$$= 10/\alpha - 3$$

Letting $\alpha = 1/(m + 1)$,

$$t_1(R) = 10(m + 1) - 3$$
$$= 10(m) + 7$$

This allows the reduction of STM as

$$
\begin{aligned}
t_1(STM) &= t(send) + t_1(P) + t(receive) + \\
&\quad t(send) + t_1(Q) + t(receive) + \\
&\quad t(send) + t_1(R) + t(receive) \\
&= 1 + 8 + 1 + 1 + 10 + 1 + \\
&\quad 1 + [10(m) + 7] + 1 \\
&= 10(m) + 31
\end{aligned}
$$

Therefore,

$$
\begin{aligned}
t_1(N) &= 3 + [8(m) + 24] \\
&= 8(m) + 27
\end{aligned}
$$

where m is an externally supplied parameter to the analysis.

CHAPTER VII. CONCLUSION

This research presents a top-down recursive method for analyzing SMD Petri net representations of data flow programs. The state machine decomposition of SMD Petri nets is similar to finding all the elementary circuits of a directed graph which is known to be an $O[(n + e)(c + 1)]$ problem [Johnson 1975] when the graph contains n nodes (transitions), e edges (arcs), and c circuits (state machines). The approach described in this presentation is significant in that

1) it does allow an approximation to be made for the execution time of data flow programs under the assumptions of infinite resources, thereby providing an approximate lower bound on execution time, and furthermore
2) this approach significantly reduces the time for analysis due to the nodal reductions to a problem of

$$O[(n_1 + e_1)\cdot(c_1 + 1) + (n_2 + e_2)\cdot(c_2 + 1) + \ldots + (n_x + e_x)\cdot(c_x + 1]$$

where x nodal reductions are made. For large programs, this may be a significant savings since $\Sigma_i n_i \simeq n$, $\Sigma_i e_i \simeq e$, and $\Sigma_i c_i << c$.

This approach yields an approximation of the lower

bound on the computation time of a program. Any inaccuracies are due primarily to three factors:

1) the assumption that all inputs must be available before the node may execute and all outputs are released simultaneously (Constraint 2 from Chapter IV),
2) the delay between any two stream values is constant, in particular $d_{1,2} = d_{last,eos} = p$, and
3) reduced nodes carry only "worst" case attributes that when used to characterize the influence on enclosing nodes may yield unobtainable results; e.g., a maximum value for t_2 used with a minimum value for n for a node internal to a streamed computation.

One of the original goals of this research was to derive a methodology that could be totally automated; i.e., given a program, an automated analysis would produce timing equations. Clearly this would be of great value in the analysis of program bottlenecks in highly concurrent systems and can be done in many cases. More research is needed to determine whether this is feasible in some situations; e.g., the recognition of streamed recursive procedures where only "scalar" connections relate input streams to output streams or the recognition of the proper context in which streamed

conditions are found. In such situations a compromise between total automation and hand-analysis may be necessary. This dissertation has provided, however, important initial steps in applying Petri net analysis of the execution of programs on several types of data flow machines with due concern given to reducing the time required for such an analysis. This concern is apparent in the methodology of recursive nodal reductions.

Since a node, corresponding to a high level construct, needs to be analyzed only once, this method is expected to provide a cost-effective technique for studying parallelism while relating the structure of a program to the degree of exhibited parallelism. An appropriate representation of the timing equations would also yield information useful in the determination of the relevancy of optimization efforts. Other potential applications concern comparisons between analogous programs written for different data flow architectures.

Suggestions for further research fall into three categories:

Category I: Improvements on methodology

1) develop methods for automatic recognition of the context of various constructs; e.g., conditionals, recursive streamed procedures,

2) determine when nodal reductions might not be appropriate thereby allowing more accurate resultant timing equations (at, perhaps, the expense of a more complicated analysis),
3) develop methods for detection of secondary, tertiary, etc., influences upon the periodic behavior of streamed computations thereby allowing more accurate approximations of the delay between any two successive tokens when desirable,
4) extend the methodology to other high level constructs; e.g., the FORALL construct, and
5) develop representation techniques for presenting the timing equations which facilitate the detection of the secondary, etc., program bottlenecks for the determination of the relevancy of optimization efforts.

Category II: Applications of methodology

1) develop approaches that facilitate the evaluation of different representations of an algorithm for a specific architecture,
2) develop approaches that allow optimization of programs, and
3) apply the methodology (with appropriate modifications) to other architectures.

Category III: Language development

The language used in this dissertation for describing computation on highly parallel architectures is not intended to be actually used by programmers. The current area of work in functionally applicative languages [Ackerman 1979, Allan and Oldehoeft 1980, Arvind, et al. 1978, Backus 1978, Chamberlin 1971, Dennis 1974, Friedman and Wise 1978, Sharp 1980, Tesler and Enea 1968, Turner 1979, Treleaven 1979, Weng 1979] holds high promise of producing acceptable languages which more naturally expose parallelism. The methodology suggested in this dissertation should equally apply for programs written in these languages.

Other related areas for research include:

1) applying the results of this research as it relates to program performance to the design of future high level languages that allow the natural expression of parallelism, and
2) applying a similar approach to the analysis of the level of demand for general-purpose and/or specialized processors as the program moves through various phases thereby yielding pertinent information as to the utilization of resources.

ACKNOWLEDGEMENTS

I wish to thank at this time my major professor, Dr. Arthur E. Oldehoeft, for the many hours he has spent in directing my research. His insights and assistance have proven to be invaluable. I also appreciate the help and time the other members of my committee, Professors Malay Ghosh, Dennis G. Kafura, Robert M. Stewart, and Roy J. Zingg, have provided me. The assistance of others, especially that of Steve Allan, Pete Maurer, and Shari Thoreson, has made a lot of this work possible.

I am very thankful for the support, both financial and emotional, my parents, Dr. and Mrs. Warren A. Jennings, have provided me through the years. My son, Jeremy, and my wife, Mary, have in many ways, perhaps unknown to them, given me the incentive to continue. I am very grateful for their love and concern. My many friends and colleagues in Ames have made this one of the most meaningful periods in my life and I am very thankful for them.

I also wish to thank Mary for her patience and help in typing this dissertation.

This work was supported in part by the National Science Foundation and by the Science and Humanities Research Institute of Iowa State University.

REFERENCES

Ackerman, W. B. 1979. Data Flow Languages. Proceedings of AFIPS, NCC 48:1Ø87-1Ø95.

Ackerman, W. B. and J. B. Dennis. 1978. VAL--A Value Oriented Algorithmic Language. Preliminary Reference Manual. Laboratory for Computer Science, M.I.T., Cambridge, MA.

Agerwala, T. 1975. Towards a Theory for the Analysis and Synthesis of Systems Exhibiting Concurrency. Ph.D. Thesis. Johns Hopkins University, Baltimore, MD.

Agerwala, T. 1979. Putting Petri Nets to Work. Computer 12, No. 12:85-94.

Allan, S. J. and A. E. Oldehoeft. 198Ø. A Flow Analysis Procedure for the Translation of High-Level Languages to a Data Flow Language. IEEE Transactions on Computers c-29:826-831.

Arvind and K. P. Gostelow. 1977. A Computer Capable of Exchanging Processor Elements for Time. Proceedings of the 1977 IFIP Congress 1977:849-853.

Arvind, K. P. Gostelow and W. Plouffe. 1978. An Asynchronous Programming Language and Computing Machine. Technical Report TR-114a. Computer Science Department, University of California, Irvine, CA.

Backus, J. 1978. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. Comm. ACM 21, No. 8:613-641.

Brock, J. D. and L. B. Montz. 1979. Translation and Optimization of Data Flow Programs. Proceedings of the 1979 International Conference on Parallel Processing 1979:46-54.

Chamberlin, D. D. 1971. Parallel Implementation of a Single-Assignment Language. Ph.D. Thesis. Stanford University.

Davis, A. L. 1978. The Architecture and System Methodology of DDM1: A Recursively Structured Data Driven Machine. Proceedings of the Fifth Symposium on Computer Architecture 5:21Ø-215.

Dennis, J. B. 1974. First Version of a Data Flow Procedure Language. Computation Structures Group Memo 93-1, Project MAC. Laboratory for Computer Science, M.I.T., Cambridge, MA.

Dennis, J. B. and D. P. Misunas. 1975. A Preliminary Architecture for a Basic Data Flow Processor. The 2nd Annual Symposium on Computer Architecture. (ACM-SIGARCH 3, No. 4:26-132).

Dennis, J. B. and K. Weng. 1979. An Abstract Implementation for Concurrent Computation with Streams. Proceedings of the 1979 International Conference on Parallel Processing 1979:35-45.

Friedman, D. P. and D. S. Wise. 1978. Aspects of Applicable Programs for Parallel Processing. IEEE Transactions on Computers c-27:289-296.

Gostelow, K. P. and R. E. Thomas. 1980. Performance of a Simulated Dataflow Computer. IEEE Transactions on Computers c-29:905-919.

Johnson, D., et al. 1980. Automatic Partitioning of Programs in Multiprocessor Systems. To appear in the Proceedings of COMPCON '80 (Spring):175-178.

Johnson, D. B. 1975. Finding All Elementary Circuits of a Directed Graph. SIAM Journal on Computing 4, No. 1:77-84.

Lee, R. B. 1980. Empirical Results on the Speed, Efficiency, Redundancy, and Quality of Parallel Computations. Proceedings of the 1980 International Conference on Parallel Processing 1980:91-100.

Mago, G. 1980. A Cellular Computer Architecture for Functional Programming. Proceedings of COMPCON '80 (Spring):179-187.

Martin, D. E. and G. Estrin. 1967. Models of Computational Systems - Cyclic to Acyclic Graph Transformations. IEEE Transactions on Electronic Computers ec-16, No. 1:70-79.

Martin, D. E. and G. Estrin. 1969. Path Length Computations on Graph Models of Computations. IEEE Transactions on Computers c-18,6:530-536.

Morris, D. and P. C. Treleaven. 1975. A Stream Processing Network. SIGPLAN Notices 10, No. 3:107-112.

Morrison, J. P. 1978. Data Stream Linkage Mechanism. IBM Systems Journal 17, No. 4:383-407.

Oldehoeft, A. E., S. Allan, S. Thoreson, C. Retnadhas and R. J. Zingg. 1978. Translation of High Level Programs to Data Flow and Their Execution on a Feedback Interpreter. Technical Report 78-2. Computer Science Department, Iowa State University, Ames, IA.

Oldehoeft, A. E., R. J. Zingg and C. Retnadhas. 1979. Measurement of Parallelism in Computer Programs through Analysis of Program Graphs. Proceedings of the First European Conference on Parallel and Distributed Processing: 1979.

Peterson, J. L. 1977. Petri Nets. ACM Computing Surveys 9, No. 3:223-252.

Plas, A., G. Durrieu, O. Gelly and J. C. Syre. 1976. LAU System Architecture: A Parallel Data Driven Processor Based on Single Assignment. Proceedings of the 1976 International Conference on Parallel Processing 1976:293-302.

Ramamoorthy, C. V. and G. S. Ho. 1980. Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets. IEEE Transactions on Software Engineering se-6:440-449.

Ramchandani, C. 1974. Analysis of Asynchronous Concurrent Systems by Petri Nets. Technical Report TR-120, Project MAC. Laboratory for Computer Science, M.I.T., Cambridge, MA.

Retnadhas, C. 1978. Compile Time Prediction of Intrinsic Parallelism in High Level Programs. Ph.D. Thesis. Iowa State University, Ames, IA.

Retnadhas, C. 1979. Execution Time Behavior of Certain High Level Constructs on a Feedback Data Flow Architecture. Proceedings of the 1979 COMPSAC Conference 1979:789-793.

Rumbaugh, J. 1977. A Data Flow Multiprocessor. IEEE Transactions on Computers c-26:138-146.

Sharp, J. A. 1980. Data Oriented Program Design. SIGPLAN Notices 15, No. 9:44-56.

Tesler, L. G. and H. J. Enea. 1968. A Language Design for Concurrent Processes. Proceedings of AFIPS, SJCC 32:402-408.

Treleaven, P. C. 1979. Exploiting Program Concurrency in Computing Systems. Computer 12, No. 1:42-50.

Turner, D. A. 1979. A New Implementation Technique for Applicative Languages. Software--Practice and Experience 9:31-49.

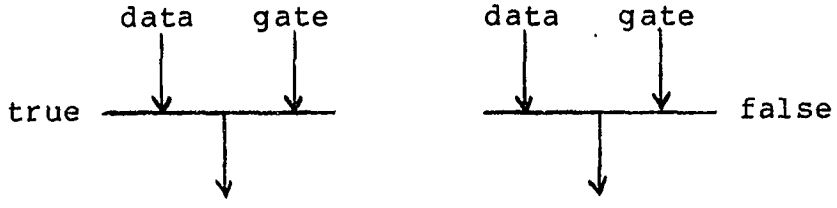
Watson, I. and J. Gurd. 1979. A Prototype Data Flow Computer with Token Labelling. Proceedings of AFIPS, NCC 48:623-628.

Weng, K. 1975. Stream-Oriented Computations in Recursive Data Flow Schemas. Technical Report TR-68, Project MAC. Laboratory for Computer Science, M.I.T., Cambridge, MA.

Weng, K. 1979. An Abstract Implementation for a Generalized Data Flow Language. Technical Report TR-228, Project MAC. Laboratory for Computer Science, M.I.T., Cambridge, MA.

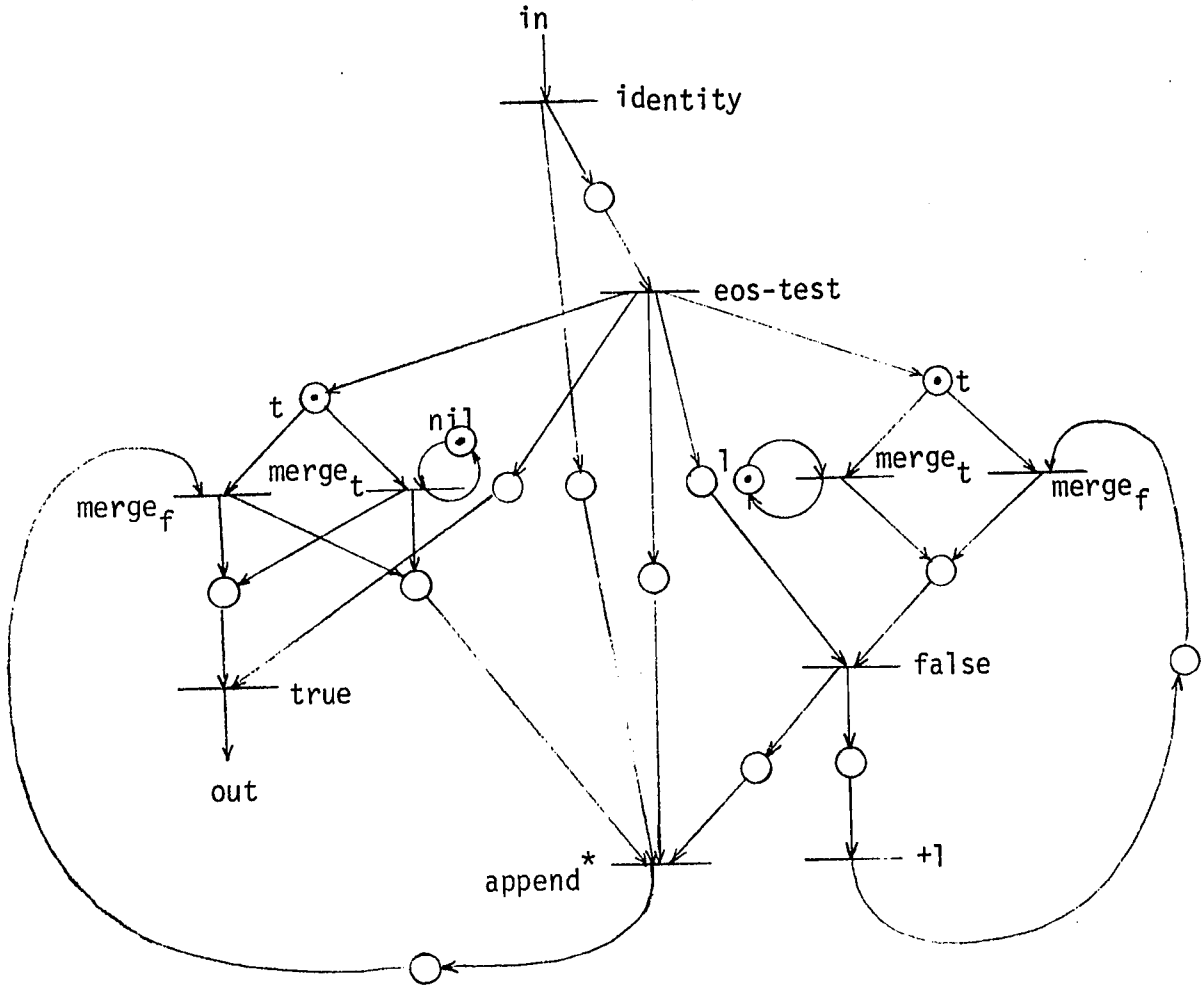
APPENDIX A. HIGH LEVEL TEMPLATES

This appendix presents the templates for the pre-analyzed high level operations. The majority of these operations are stream operations. These templates are those found in the ISU data flow compiler. Switch operations in this system are implemented using "true" and "false" gating operations. A "true" operation takes a data value and a gate value as input operands. Should the gate value be true then the data value is simply passed on. A false value destroys the data value. The "false" operation functions similarly. The switch operation may therefore be modeled as:



with the understanding that under certain conditions the operation may not produce an output token. It is possible in the ISU data flow system to embed some of the gating in other instructions. Where this occurs, it will be noted. As mentioned earlier, the underlying semantics provide a deterministic firing sequence that allows for the specification of attributes.

BUF



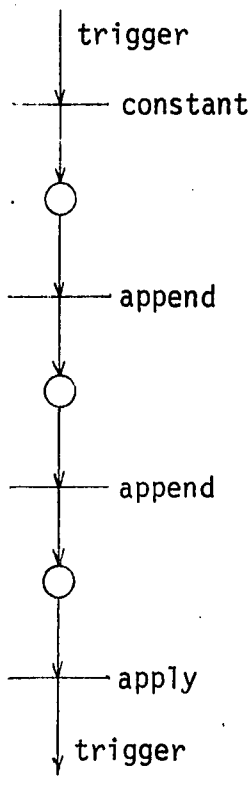
$t_1 = 3$

IP = 4

---

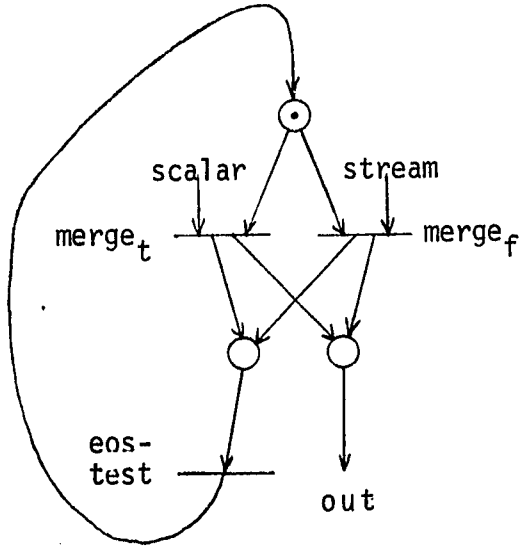
* - with true gating

CALL



$t_1 = 4$

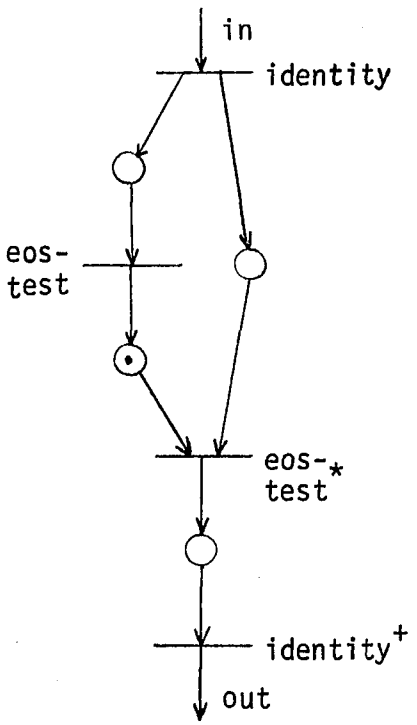
CONS



$t_1 = t_2 = 1$

IP = 2

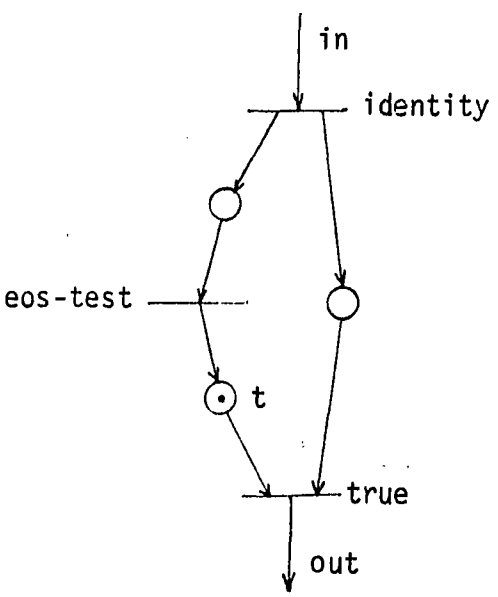
EMPTY



$t_1 = 3$

IP = 3

FIRST



$t_1 = 2$

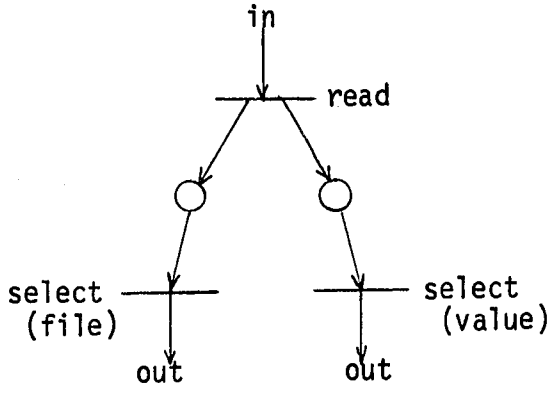
IP = 3

---

* - with true gating

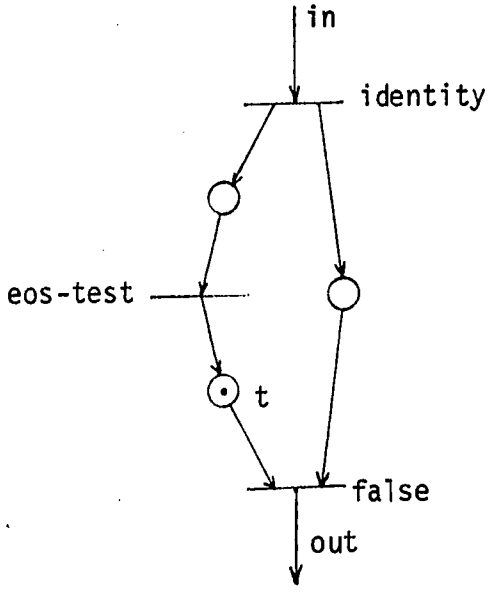
\+ - used to convert boolean value to gate value

INPUT (scalar)



$t_1 = 2$

REST

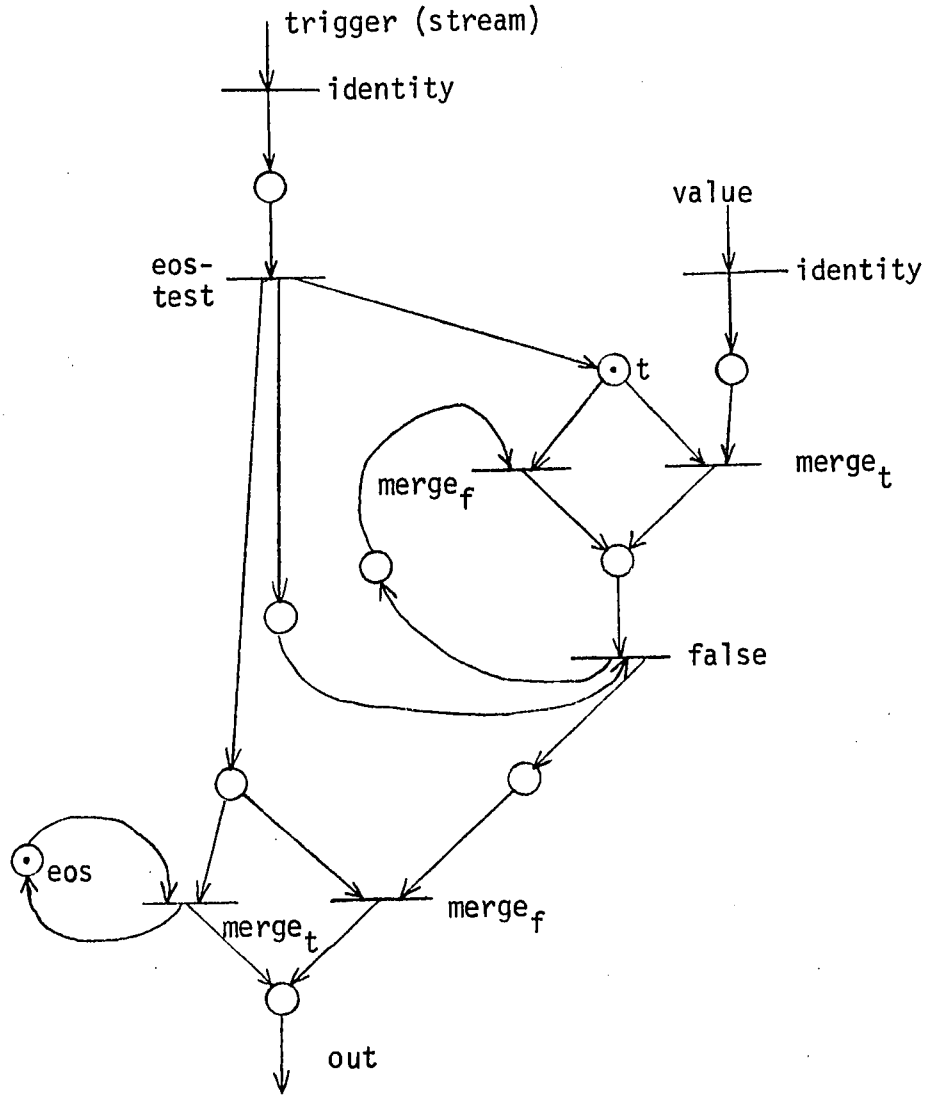


$t_1 = 2 + \max(p,3)$

$t = 2$

$IP = 3$

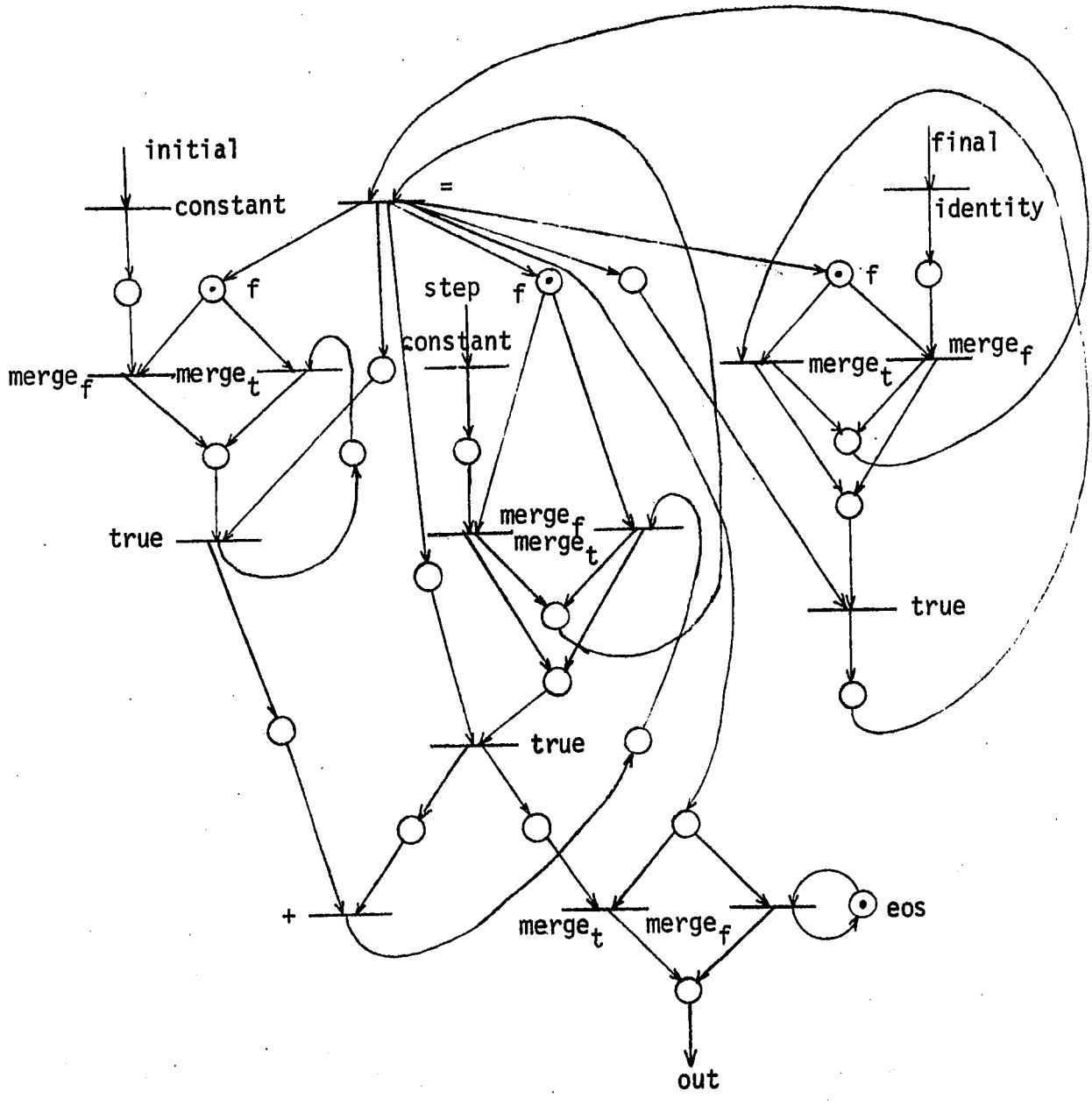
REPL



$t_1 = t_2 = 4$

IP = 3

SCREATE



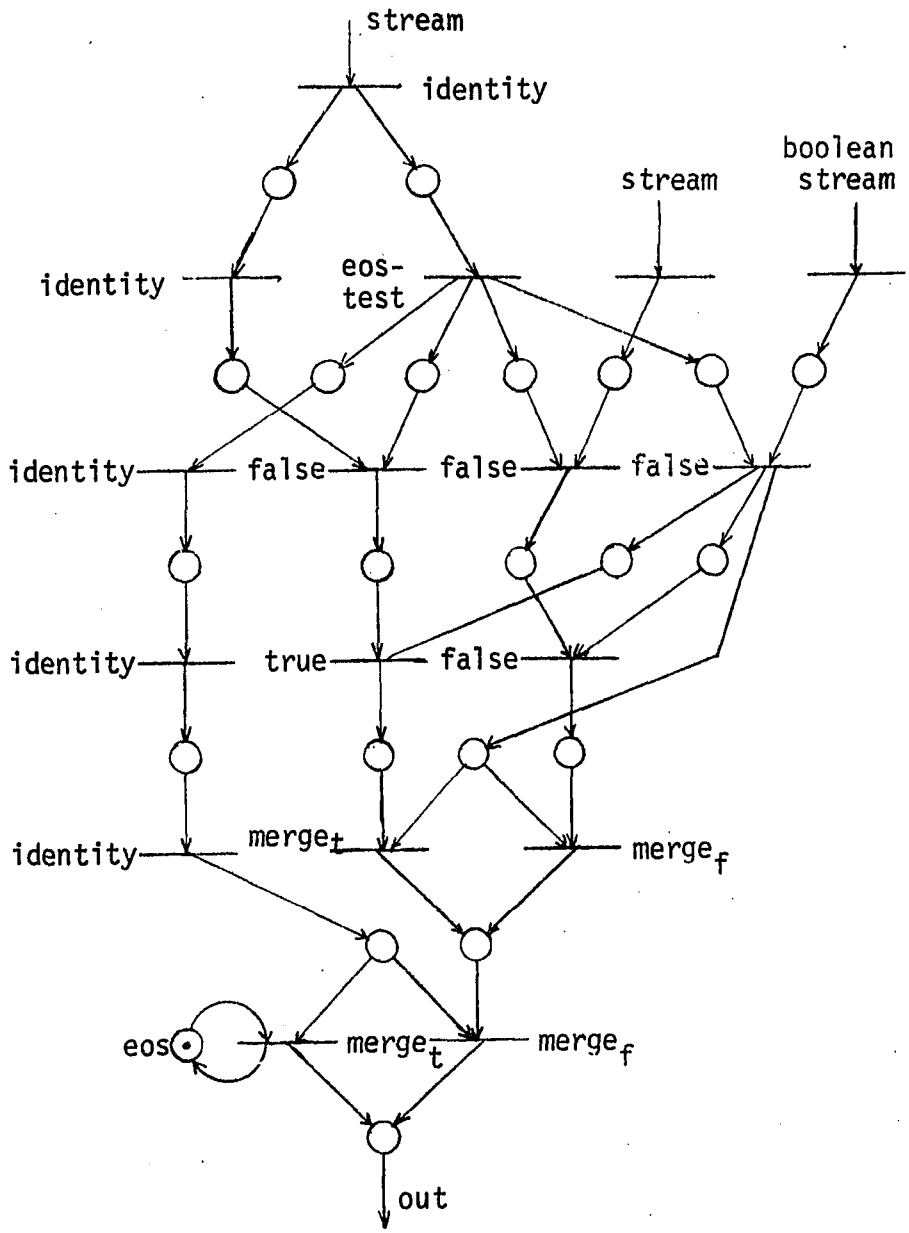
$t_1 = 5$

IP = 4*

---

* - $d_{last,eos} = 3$

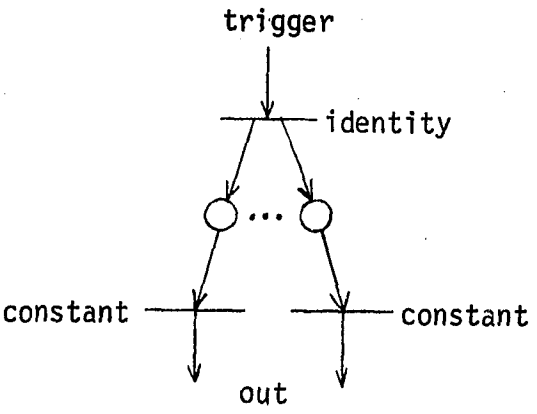
SELECT



$t_1 = t_2 = 6$

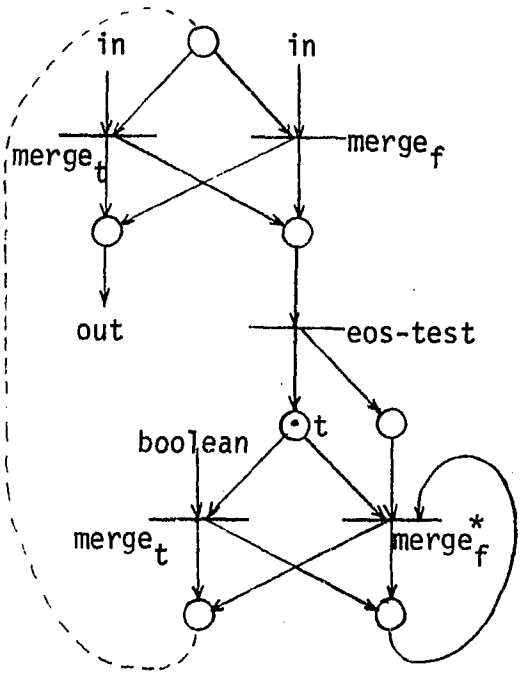
IP = 2

START



$t_1 = 2$

STR-MERGE



$t = 1^+$

$t = 1$

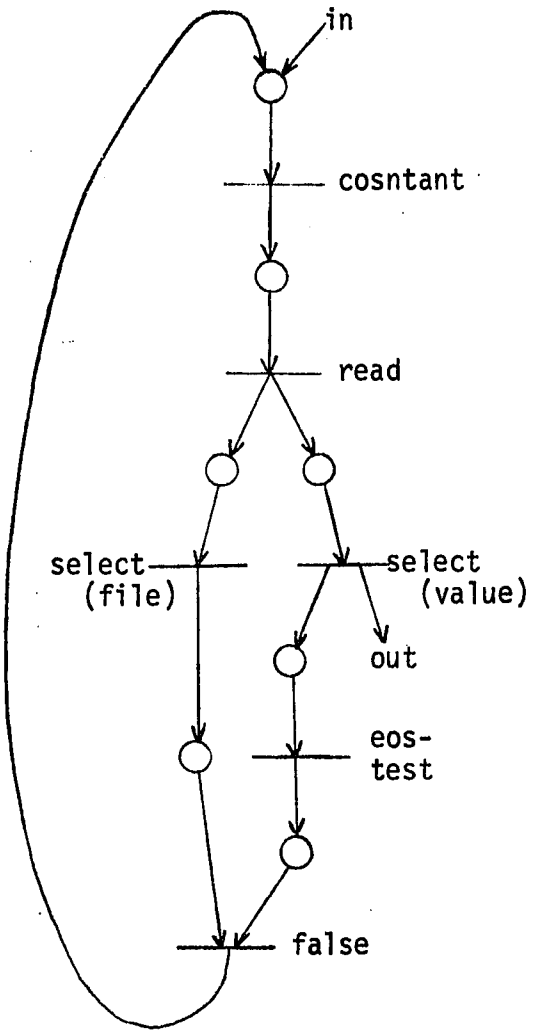
$IP = 3$

---

* - with false gating

\+ - since this template is used only to terminate a condition, the boolean arrives sooner and hence this value results.

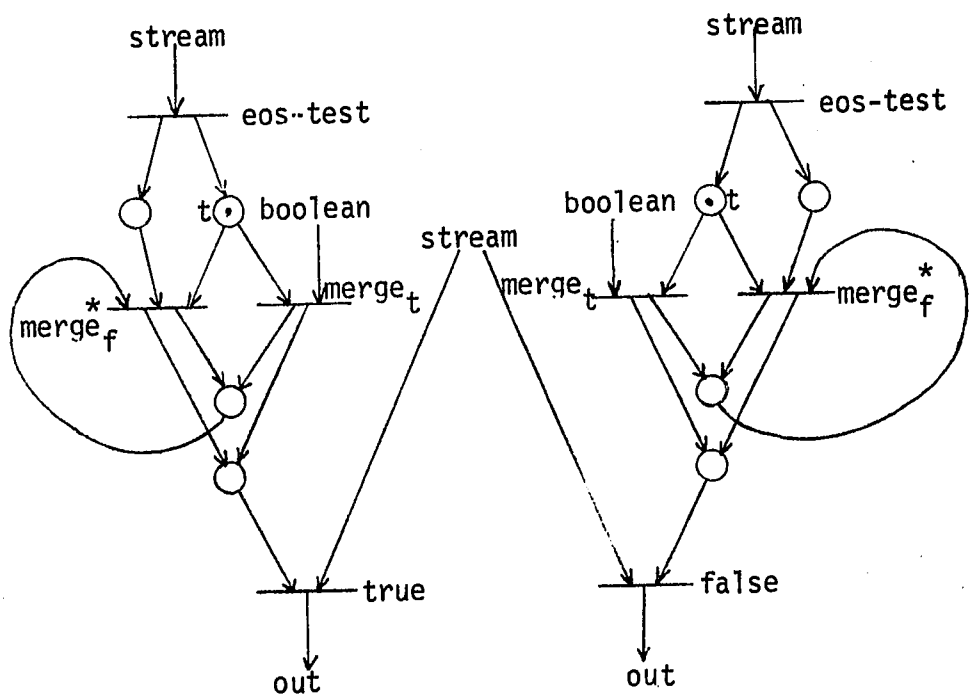
STR-INPUT



$t_1 = 3$

IP = 5

STR-SWITCH



$t_1 = 2$

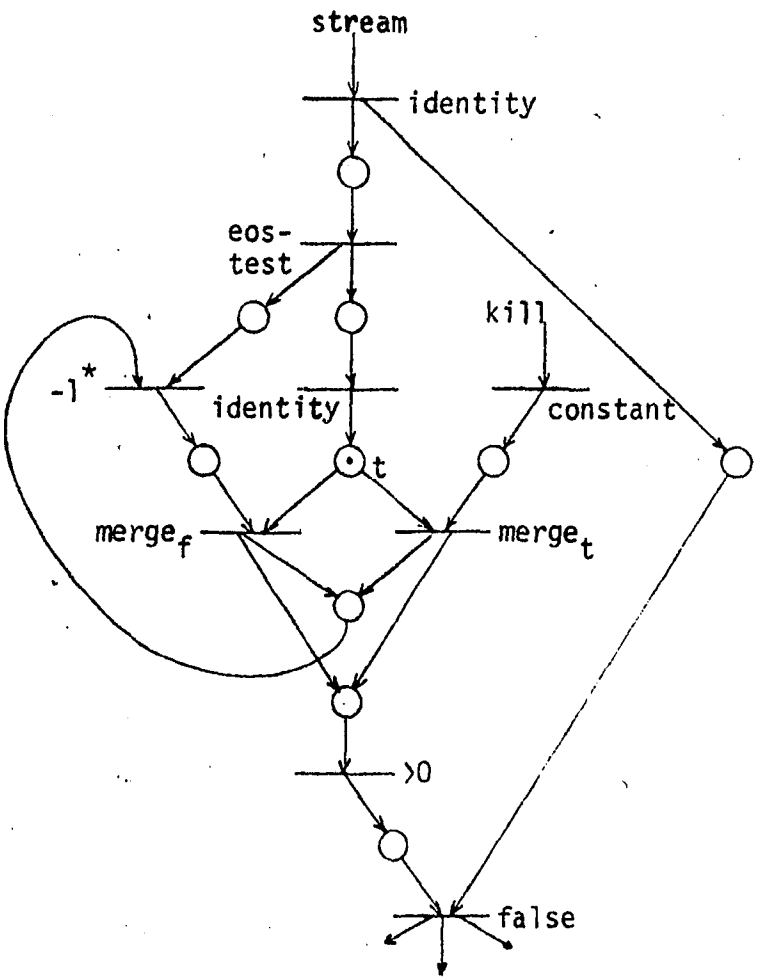
$t_2 = 1$

IP = 3

---

* - with false gating

SUBSTM



(continued)

$t_1 = 7 + k \quad \max[3, p]$

$t_2 = 5$

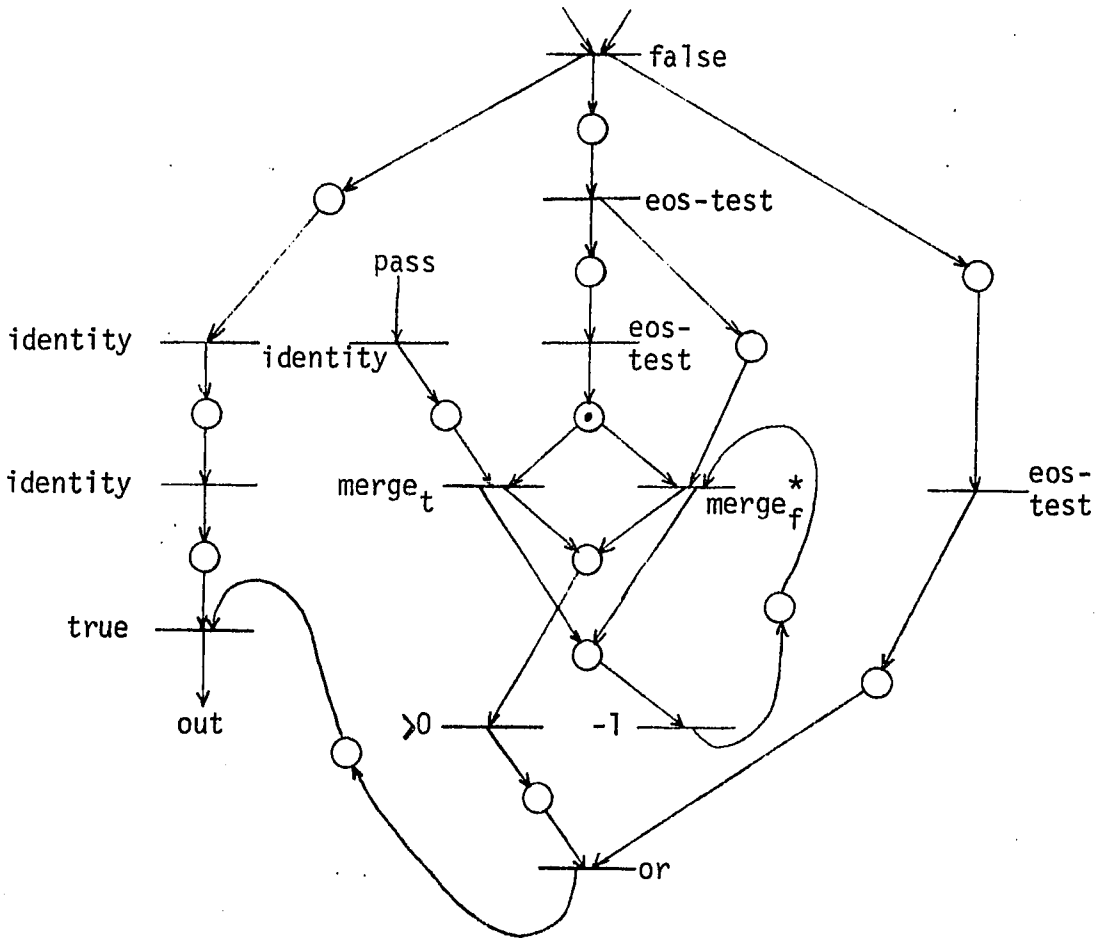
IP = 3

---

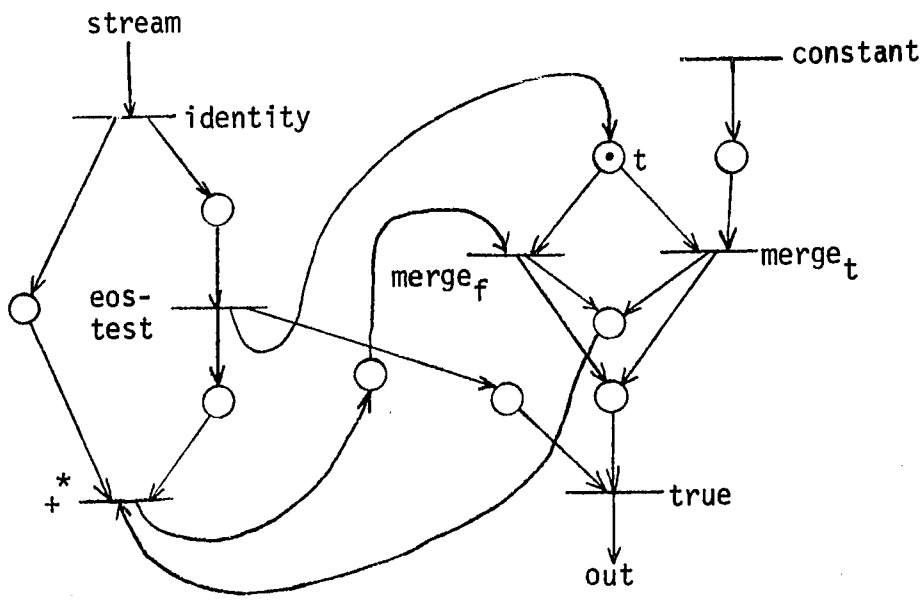
* - with false gating

\+ - should the scalars arrive sooner, this value may be reduced by two time steps.

SUBSTM (continued)



SUM (PROD)

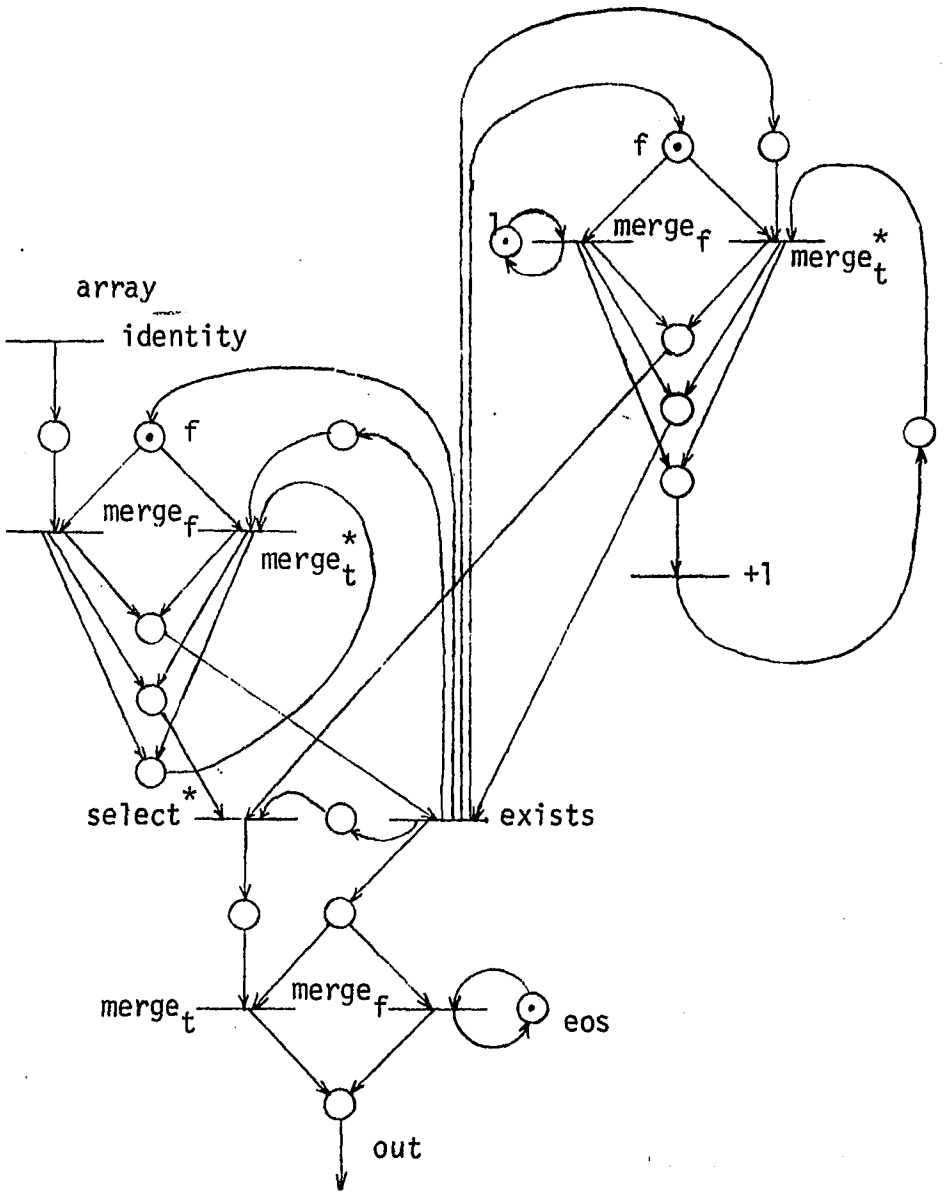


$t_1 = 2 + m \quad \max(3,p)$

---

* - with false gating

UNBUF



$t_1 = 5$

IP = 4

---

* - with true gating

APPENDIX B. TEST PROGRAMS

This appendix contains the ISU data flow versions of the programs analyzed in Chapter V: HYSL, NEXP, TRIG, SSUM, and MULT. For each program (in the specified order) is found

1) the high level language version,
2) the assembled code, and
3) the JCL and input records to compile, assemble, and simulate the program.

```
PROCEDURE HYSL BEGIN
  REAL ARRAY AP(1:11), ARHO(1:11), AX(1:12);
  INTEGER TS2, ENDCYCLE, MAXZ, I;
  REAL STREAM P, RHO, X, ACC, Q;
  FILE INFILE;
  INPUT MAXZ, ENDCYCLE  FILE=INFILE FORMAT=(I(2),I(2));
  TS2 := 0;
  BUF SCREATE(IN(1,MAXZ,1)), AP;
  BUF SCREATE(IN(1,MAXZ,1)), ARHO;
  BUF SCREATE(IN(0,MAXZ,1)), AX;
  WHILE TS2 < ENDCYCLE DO
    P := UNBUF(IN(AP));
    X := UNBUF(IN(AX));
    RHO := UNBUF(IN(ARHO));
    ACC := CONS(IN(2*FIRST(IN(P))/FIRST(IN(RHO)),
                    REPL(IN(2,P))*P/RHO));
    X := X + ACC * REPL(IN(2,ACC));
    RHO := RHO * (SUBSTM(IN(X,1,MAXZ))-
                  SUBSTM(IN(X,0,MAXZ)));
    Q := RHO * REPL(IN(0.5,RHO));
    P := SELECT(IN(REPL(IN(0.0,Q)), Q,
                    ABS(IN(Q))<REPL(IN(0.0,Q))));
    BUF P, AP;
    BUF X, AX;
    BUF RHO, ARHO;
    TS2 := TS2 + 1
    END
  END
```

```
PROC HYSL
      0 ID (T=S,R='NIL')(T=S,D=1.1,51.1,49.1,31.1,29.1,11.1,9.1,8.1)
      1 CONS (T=S)(T=F,R='INFILE',C=C) (T=F,D=2.1)
      2 READ (T=F)(R='I(2)',T=C,C=C)(T=S,D=3.1,4.1)
      3 SELECT (T=S)(R=1,T=I,C=C)(T=F,D=5.1)
      4 SELECT (T=S)(R=2,T=I,C=C)(T=I,D=10.1,75.2,50.1,30.1)
      5 READ (T=F)(R='I(2)',T=C,C=C)(T=S,D=6.1,7.1)
      6 SELECT (T=S)(R=1,T=I,C=C)(T=F,D=)
      7 SELECT (T=S)(R=2,T=I,C=C)(T=I,D=71.2)
      8 CONS (T=S)(T=I,R=0,C=C)(T=I,D=70.2)
      9 CONS (T=S)(T=I,C=C,R=1)(T=I,D=12.2)
     10 ID (T=I)(T=I,D=18.2)
     11 CONS (T=S)(T=I,C=C,R=1)(T=I,D=14.2)
     12 MERGE (T=I)(T=I)(G=F,@=17)(T=I,D=13.1)
     13 ID (T=I,C=T)(T=I,D=12.1,16.1)
     14 MERGE (T=I)(T=I)(G=F,@=17)(T=I,D=15.1,17.1)
     15 ID (T=I,C=T)(T=I,D=16.2,19.1)
     16 + (T=I)(T=I)(T=I,D=14.1)
     17 <= (T=I)(T=I)(C=3)(T=G,D=12.0,13.1,14.0,15.1,18.0,20.1,19.0)
     18 MERGE (T=I)(T=I)(G=F,@=17)(T=I,D=17.2,20.1)
     19 MERGE (T=I)(T=I,C=C,R=524288)()(T=I,D=21.1)
     20 ID (T=I,C=T)(T=I,D=18.1)
     21 ID (T=I)(T=R,D=22.1,27.3)
     22 EOS (T=R)(C=2)(T=G,D=23.0,24.1,26.0,27.1,27.3,28.1)
     23 MERGE (T=I,C=C,R=1)(T=I)(G=T,@=22)(T=I,D=24.1)
     24 ID (T=I,C=F)(T=I,D=25.1,27.2)
     25 + (T=I)(T=I,C=C,R=1)(T=I,D=23.2)
     26 MERGE (T=S,C=C,R='NIL')(T=S)(G=T,@=22)(T=S,D=27.1,28.1)
     27 APPEND (T=S,C=F)(T=I)(T=R,C=F)(T=S,D=26.2)
     28 ID (T=S,C=T)(T=S,D=72.2)
     29 CONS (T=S)(T=I,C=C,R=1)(T=I,D=32.2)
     30 ID (T=I)(T=I,D=38.2)
     31 CONS (T=S)(T=I,C=C,R=1)(T=I,D=34.2)
     32 MERGE (T=I)(T=I)(G=F,@=37)(T=I,D=33.1)
     33 ID (T=I,C=T)(T=I,D=32.1,36.1)
     34 MERGE (T=I)(T=I)(G=F,@=37)(T=I,D=35.1,37.1)
     35 ID (T=I,C=T)(T=I,D=36.2,39.1)
```

```
36 + (T=I)(T=I)(T=I,D=34.1)
37 <= (T=I)(T=I)(C=3)(T=G,D=32.0,33.1,34.0,35.1,38.0,40.1,39.0)
38 MERGE (T=I)(T=I)(G=F,@=37)(T=I,D=37.2,40.1)
39 MERGE (T=I)(T=I,C=C,R=524288)()(T=I,D=41.1)
40 ID (T=I,C=T)(T=I,D=38.1)
41 ID (T=I)(T=R,D=42.1,47.3)
42 EOS (T=R)(C=2)(T=G,D=43.0,44.1,46.0,47.1,47.3,48.1)
43 MERGE (T=I,C=C,R=1)(T=I)(G=T,@=42)(T=I,D=44.1)
44 ID (T=I,C=F)(T=I,D=45.1,47.2)
45 + (T=I)(T=I,C=C,R=1)(T=I,D=43.2)
46 MERGE (T=S,C=C,R='NIL')(T=S)(G=T,@=42)(T=S,D=47.1,48.1)
47 APPEND (T=S,C=F)(T=I)(T=R,C=F)(T=S,D=46.2)
48 ID (T=S,C=T)(T=S,D=74.2)
49 CONS (T=S)(T=I,C=C,R=1)(T=I,D=52.2)
50 ID (T=I)(T=I,D=58.2)
51 CONS (T=S)(T=I,C=C,R=0)(T=I,D=54.2)
52 MERGE (T=I)(T=I)(G=F,@=57)(T=I,D=53.1)
53 ID (T=I,C=T)(T=I,D=52.1,56.1)
54 MERGE (T=I)(T=I)(G=F,@=57)(T=I,D=55.1,57.1)
55 ID (T=I,C=T)(T=I,D=56.2,59.1)
56 + (T=I)(T=I)(T=I,D=54.1)
57 <= (T=I)(T=I)(C=3)(T=G,D=52.0,53.1,54.0,55.1,58.0,60.1,59.0)
58 MERGE (T=I)(T=I)(G=F,@=57)(T=I,D=57.2,60.1)
59 MERGE (T=I)(T=I,C=C,R=524288)()(T=I,D=61.1)
60 ID (T=I,C=T)(T=I,D=58.1)
61 ID (T=I)(T=R,D=62.1,67.3)
62 EOS (T=R)(C=2)(T=G,D=63.0,64.1,66.0,67.1,67.3,68.1)
63 MERGE (T=I,C=C,R=1)(T=I)(G=T,@=62)(T=I,D=64.1)
64 ID (T=I,C=F)(T=I,D=65.1,67.2)
65 + (T=I)(T=I,C=C,R=1)(T=I,D=63.2)
66 MERGE (T=S,C=C,R='NIL')(T=S)(G=T,@=62)(T=S,D=67.1,68.1)
67 APPEND (T=S,C=F)(T=I)(T=R,C=F)(T=S,D=66.2)
68 ID (T=S,C=T)(T=S,D=73.2)
70 MERGE (T=I)(T=I)(G=F,@=69)(T=I,D=69.1,76.1)
71 MERGE (T=I)(T=I)(G=F,@=69)(T=I,D=69.2,77.1)
72 MERGE (T=S)(T=S)(G=F,@=69)(T=S,D=78.1,230.1)
73 MERGE (T=S)(T=S)(G=F,@=69)(T=S,D=79.1,231.1)
```

```
74 MERGE (T=S)(T=S)(G=F,@=69)(T=S,D=80.1,232.1)
75 MERGE (T=I)(T=I)(G=F,@=69)(T=I,D=81.1)
69 < (T=I)(T=I)(C=6)(T=G,D=70.0,232.1,231.1,230.1,81.1,80.1,79.
   78.1,77.1,76.1,75.0,74.0,73.0,72.0,71.0)
76 ID (T=I,C=T)(T=I,D=112.1,229.1,185.1,178.1,170.1,149.1,130.1
   122.1)
77 ID (T=I,C=T)(T=I,D=71.1)
78 ID (T=S,C=T)(T=S,D=82.1)
79 ID (T=S,C=T)(T=S,D=89.1)
80 ID (T=S,C=T)(T=S,D=96.1)
81 ID (T=I,C=T)(T=I,D=75.1,148.1,129.1)
82 ID (T=S)(T=S,D=83.2)
83 MERGE (T=S,C=T)(T=S)(G=F,@=86)(T=S,D=83.1,86.1,87.1)
84 MERGE (T=I,C=T)(T=I,C=C,R=1)(G=F,@=86)(T=I,D=85.1,86.2,87.2)
85 + (T=I)(T=I,C=C,R=1)(T=I,D=84.1)
86 EXISTS (T=S)(T=I)(C=2)(T=G,D=83.0,83.1,84.0,84.1,87.1,87.2,
   88.0)
87 SELECT (T=S,C=T)(T=I,C=T)(T=R,D=88.1)
88 MERGE (T=R)(T=R,C=C,R=524288)()(T=R,D=103.1,117.2,111.1)
89 ID (T=S)(T=S,D=90.2)
90 MERGE (T=S,C=T)(T=S)(G=F,@=93)(T=S,D=90.1,93.1,94.1)
91 MERGE (T=I,C=T)(T=I,C=C,R=1)(G=F,@=93)(T=I,D=92.1,93.2,94.2)
92 + (T=I)(T=I,C=C,R=1)(T=I,D=91.1)
93 EXISTS (T=S)(T=I)(C=2)(T=G,D=90.0,90.1,91.0,91.1,94.1,94.2,
   95.0)
94 SELECT (T=S,C=T)(T=I,C=T)(T=R,D=95.1)
95 MERGE (T=R)(T=R,C=C,R=524288)()(T=R,D=128.1)
96 ID (T=S)(T=S,D=97.2)
97 MERGE (T=S,C=T)(T=S)(G=F,@=100)(T=S,D=97.1,100.1,101.1)
98 MERGE (T=I,C=T)(T=I,C=C,R=1)(G=F,@=100)(T=I,D=99.1,100.2,101
99 + (T=I)(T=I,C=C,R=1)(T=I,D=98.1)
100 EXISTS (T=S)(T=I)(C=2)(T=G,D=97.0,97.1,98.0,98.1,101.1,101.2
    102.0)
101 SELECT (T=S,C=T)(T=I,C=T)(T=R,D=102.1)
102 MERGE (T=R)(T=R,C=C,R=524288)()(T=R,D=107.1,168.1,118.2)
103 ID (T=R)(T=R,D=104.1,105.1)
```

```
104 EOS (T=R)(C=1)(T=G,D=105.1)
105 ID (T=R,C=T,G=T,@=104)(T=R,D=106.2)
106 * (T=I,R=2,C=C)(T=R)(T=R,D=110.1)
107 ID (T=R)(T=R,D=108.1,109.1)
108 EOS (T=R)(C=1)(T=G,D=109.1)
109 ID (T=R,C=T,G=T,@=108)(T=R,D=110.2)
110 / (T=R)(T=R)(T=R,D=119.1)
111 ID (T=R)(T=R,D=113.1)
112 CONS (T=I)(T=I,C=C,R=2)(T=I,D=114.1)
113 EOS (T=R)(C=1)(T=G,D=114.0,115.1,116.0)
114 MERGE (T=I)(T=I)(G=T,@=113)(T=I,D=115.1)
115 ID (T=I,C=F)(T=I,D=114.2,116.2)
116 MERGE (T=I,C=C,R=524288)(T=I)()(T=I,D=117.1)
117 * (T=I)(T=R)(T=R,D=118.1)
118 / (T=R)(T=R)(T=R,D=119.2)
119 MERGE (T=R)(T=R)(G=T,@=120)(T=R,D=120.1,127.1,121.1)
120 EOS (T=R)(C=1)(T=G,D=119.0)
121 ID (T=R)(T=R,D=123.1)
122 CONS (T=I)(T=I,C=C,R=2)(T=I,D=124.1)
123 EOS (T=R)(C=1)(T=G,D=124.0,125.1,126.0)
124 MERGE (T=I)(T=I)(G=T,@=123)(T=I,D=125.1)
125 ID (T=I,C=F)(T=I,D=124.2,126.2)
126 MERGE (T=I,C=C,R=524288)(T=I)()(T=I,D=127.2)
127 * (T=R)(T=I)(T=R,D=128.2)
128 + (T=R)(T=R)(T=R,D=131.1,213.1,150.1)
129 ID (T=I)(T=I,D=143.1)
130 CONS (T=I)(T=I,C=C,R=1)(T=I,D=141.1)
131 ID (T=R)(T=R,D=135.1,136.1)
132 - (T=I,C=F)(T=I,C=C,R=1)(T=I,D=141.2)
133 > (T=I)(T=I,C=C,R=0)(T=G,D=136.1)
134 ID (T=G)(C=1)(T=G,D=141.0)
135 EOS (T=R)(T=G,D=132.1,134.1,D)
136 ID (T=R,C=F)(T=R,D=137.1,138.1,146.1)
137 EOS (T=R)(T=G,D=142.1,D,143.2)
138 ID (T=R)(T=R,D=139.1)
139 ID (T=R)(T=R,D=140.1)
140 ID (T=R,C=T)(T=R,D=167.1)
```

```
141 MERGE (T=I)(T=I)(G=T,@=134)(T=I,D=132.1,133.1)
142 ID (T=G)(C=1)(T=G,D=143.0)
143 MERGE (T=I)(T=I,C=F)(G=T,@=142)(T=I,D=144.1,145.1)
144 - (T=I)(T=I,C=C,R=1)(T=I,D=143.2)
145 > (T=I)(T=I,C=C,R=0)(T=B,D=147.1)
146 EOS (T=R)(T=B,D=147.2)
147 | (T=B)(T=B)(T=G,D=140.1)
148 ID (T=I)(T=I,D=162.1)
149 CONS (T=I)(T=I,C=C,R=0)(T=I,D=160.1)
150 ID (T=R)(T=R,D=154.1,155.1)
151 - (T=I,C=F)(T=I,C=C,R=1)(T=I,D=160.2)
152 > (T=I)(T=I,C=C,R=0)(T=G,D=155.1)
153 ID (T=G)(C=1)(T=G,D=160.0)
154 EOS (T=R)(T=G,D=151.1,153.1,D)
155 ID (T=R,C=F)(T=R,D=156.1,157.1,165.1)
156 EOS (T=R)(T=G,D=161.1,D,162.2)
157 ID (T=R)(T=R,D=158.1)
158 ID (T=R)(T=R,D=159.1)
159 ID (T=R,C=T)(T=R,D=167.2)
160 MERGE (T=I)(T=I)(G=T,@=153)(T=I,D=151.1,152.1)
161 ID (T=G)(C=1)(T=G,D=162.0)
162 MERGE (T=I)(T=I,C=F)(G=T,@=161)(T=I,D=163.1,164.1)
163 - (T=I)(T=I,C=C,R=1)(T=I,D=162.2)
164 > (T=I)(T=I,C=C,R=0)(T=B,D=166.1)
165 EOS (T=R)(T=B,D=166.2)
166 | (T=B)(T=B)(T=G,D=159.1)
167 - (T=R)(T=R)(T=R,D=168.2)
168 * (T=R)(T=R)(T=R,D=169.1,221.1,175.1)
169 ID (T=R)(T=R,D=171.1)
170 CONS (T=I)(T=R,C=C,R=0.5)(T=R,D=172.1)
171 EOS (T=R)(C=1)(T=G,D=172.0,173.1,174.0)
172 MERGE (T=R)(T=R)(G=T,@=171)(T=R,D=173.1)
173 ID (T=R,C=F)(T=R,D=172.2,174.2)
174 MERGE (T=R,C=C,R=524288)(T=R)()(T=R,D=175.2)
175 * (T=R)(T=R)(T=R,D=176.1,195.1,184.1,177.1)
176 ABS (T=R)(T=R,D=183.1)
177 ID (T=R)(T=R,D=179.1)
```

```
178 CONS (T=I)(T=R,C=C,R=0.0)(T=R,D=180.1)
179 EOS (T=R)(C=1)(T=G,D=180.0,181.1,182.0)
180 MERGE (T=R)(T=R)(G=T,@=179)(T=R,D=181.1)
181 ID (T=R,C=F)(T=R,D=180.2,182.2)
182 MERGE (T=R,C=C,R=524288)(T=R)()(T=R,D=183.2)
183 < (T=R)(T=R)(T=B,D=198.1)
184 ID (T=R)(T=R,D=186.1)
185 CONS (T=I)(T=R,C=C,R=0.0)(T=R,D=187.1)
186 EOS (T=R)(C=1)(T=G,D=187.0,188.1,189.0)
187 MERGE (T=R)(T=R)(G=T,@=186)(T=R,D=188.1)
188 ID (T=R,C=F)(T=R,D=187.2,189.2)
189 MERGE (T=R,C=C,R=524288)(T=R)()(T=R,D=190.1)
190 ID (T=R)(T=R,D=191.1,192.1)
191 EOS (T=R)(T=G,D=193.1,196.1,199.1,200.1.D)
192 ID (T=R)(T=R,D=193.1)
193 ID (T=R,C=F)(T=R,D=194.1)
194 ID (T=R,C=T)(T=R,D=203.1)
195 ID (T=R)(T=R,D=196.1)
196 ID (T=R,C=F)(T=R,D=197.1)
197 ID (T=R,C=F)(T=R,D=203.2)
198 ID (T=B)(T=B,D=199.1)
199 ID (T=B,C=F)(T=G,D=194.1,197.1,203.0)
200 ID (T=G)(T=G,D=201.1.D)
201 ID (T=G)(T=G,D=202.1.D)
202 ID (T=G)(T=G,D=204.0)
203 MERGE (T=R)(T=R)()(T=R,D=204.2)
204 MERGE (T=R,C=C,R=524288)(T=R)()(T=R,D=205.1)
205 ID (T=R)(T=R,D=206.1,211.3)
206 EOS (T=R)(C=2)(T=G,D=207.0,208.1,210.0,211.1,211.3,212.1)
207 MERGE (T=I,C=C,R=1)(T=I)(G=T,@=206)(T=I,D=208.1)
208 ID (T=I,C=F)(T=I,D=209.1,211.2)
209 + (T=I)(T=I,C=C,R=1)(T=I,D=207.2)
210 MERGE (T=S,C=C,R='NIL')(T=S)(G=T,@=206)(T=S,D=211.1,212.1)
211 APPEND (T=S,C=F)(T=I)(T=R,C=F)(T=S,D=210.2)
212 ID (T=S,C=T)(T=S,D=72.1)
213 ID (T=R)(T=R,D=214.1,219.3)
```

```
214 EOS (T=R)(C=2)(T=G,D=215.0,216.1,218.0,219.1,219.3,220.1)
215 MERGE (T=I,C=C,R=1)(T=I)(G=T,@=214)(T=I,D=216.1)
216 ID (T=I,C=F)(T=I,D=217.1,219.2)
217 + (T=I)(T=I,C=C,R=1)(T=I,D=215.2)
218 MERGE (T=S,C=C,R='NIL')(T=S)(G=T,@=214)(T=S,D=219.1,220.1)
219 APPEND (T=S,C=F)(T=I)(T=R,C=F)(T=S,D=218.2)
220 ID (T=S,C=T)(T=S,D=73.1)
221 ID (T=R)(T=R,D=222.1,227.3)
222 EOS (T=R)(C=2)(T=G,D=223.0,224.1,226.0,227.1,227.3,228.1)
223 MERGE (T=I,C=C,R=1)(T=I)(G=T,@=222)(T=I,D=224.1)
224 ID (T=I,C=F)(T=I,D=225.1,227.2)
225 + (T=I)(T=I,C=C,R=1)(T=I,D=223.2)
226 MERGE (T=S,C=C,R='NIL')(T=S)(G=T,@=222)(T=S,D=227.1,228.1)
227 APPEND (T=S,C=F)(T=I)(T=R,C=F)(T=S,D=226.2)
228 ID (T=S,C=T)(T=S,D=74.1)
229 + (T=I)(T=I,R=1,C=C)(T=I,D=70.1)
230 ID (T=S,C=F)(T=S,D=)
231 ID (T=S,C=F)(T=S,D=)
232 ID (T=S,C=F)(T=S,D=)
```

```
//E327HYSL  JOB  I4987,CPLTJCL,MSGLEVEL=(1,1)
//DFC  EXEC  PGM=DFC,REGION=356K,TIME=(,10),PARM=('ISA(-14K)/',
//   'CODE')
//STEPLIB  DD DSN=A.I4987.DFC.NEWCOM2,DISP=SHR
//SYSPRINT  DD  SYSOUT=A
//CODEFIL DD DSN=J.I4987.DFC.CODE1,UNIT=DISK,
//  DISP=(NEW,CATLG,DELETE),
//  DCB=(LRECL=80,BLKSIZE=800,RECFM=FB,BUFNO=1),
//  SPACE=(TRK,(10,1),RLSE)
//LASTFIL DD DSN=J.I4987.CODE.HYSL,UNIT=DISK,
//    DCB=(LRECL=80,BLKSIZE=800,RECFM=FB,BUFNO=1),
//    DISP=(NEW,CATLG,DELETE),SPACE=(TRK,(10,2),RLSE,CONTIG)
//SYSTEST  DD  SYSOUT=A
//SYSDUMP  DD SYSOUT=A
//PLIDUMP  DD  SYSOUT=A,DCB=BLKSIZE=133
//SYSIN  DD *
 PROCEDURE HYSL BEGIN
   REAL ARRAY AP(1:11), ARHO(1:11), AX(1:12);
   INTEGER TS2, ENDCYCLE, MAXZ, I;
   REAL STREAM P, RHO, X, ACC, Q;
   FILE INFILE;
   INPUT MAXZ, ENDCYCLE  FILE=INFILE FORMAT=(I(2),I(2));
   TS2 := 0;
   BUF SCREATE(IN(1,MAXZ,1)), AP;
   BUF SCREATE(IN(1,MAXZ,1)), ARHO;
   BUF SCREATE(IN(0,MAXZ,1)), AX;
   WHILE TS2 < ENDCYCLE DO
     P := UNBUF(IN(AP));
     X := UNBUF(IN(AX));
     RHO := UNBUF(IN(ARHO));
     ACC := CONS(IN(2*FIRST(IN(P))/FIRST(IN(RHO)),
                    REPL(IN(2,P))*P/RHO));
     X := X + ACC * REPL(IN(2,ACC));
     RHO := RHO * (SUBSTM(IN(X,1,MAXZ))-
                   SUBSTM(IN(X,0,MAXZ)));
     Q := RHO * REPL(IN(0.5,RHO));
     P := SELECT(IN(REPL(IN(0.0,Q)), Q,
                    ABS(IN(Q))<REPL(IN(0.0,Q))));
     BUF P, AP;
     BUF X, AX;
     BUF RHO, ARHO;
     TS2 := TS2 + 1
     END
   END

//STEP2 EXEC PGM=IEBPTPCH,COND=(0,EQ,DFC)
//SYSPRINT DD SYSOUT=A
//SYSUT1 DD DSN=J.I4987.DFC.CODE1,DISP=(OLD,DELETE)
//SYSUT2 DD SYSOUT=A
//SYSIN DD *
```

```
 PRINT MAXFLDS=1
 TITLE ITEM=('HYSL')
 RECORD FIELD=(80)
/*
//DEL DD DSN=J.I4987.CODE.HYSL,DISP=(OLD,DELETE)
//XS1 EXEC PGM=IEFBR14,COND=(0,NE,DFC)
//DD1 DD DSN=J.I4987.DFC.CODE1,DISP=(OLD,DELETE)
//ASM EXEC PGM=GO,REGION=100K,TIME=(,10),COND=(0,NE,DFC)
//* DD STATEMENTS FOR ASSEMBLER STEP
//STEPLIB DD DSN=P.I4119.ASMLOAD,DISP=SHR
//SYSIN DD DSN=J.I4987.CODE.HYSL,DISP=(OLD,DELETE)
//SYSPRINT DD SYSOUT=A
//PLIDUMP DD SYSOUT=A,DCB=BLKSIZE=133
//SYSUDUMP DD SYSOUT=A
//MARC DD DSN=J.I4987.ASM.HYSL,UNIT=DISK,DISP=(NEW,CATLG,DELETE),
//  SPACE=(TRK,(10,1),RLSE),
// DCB=(DSORG=DA,LRECL=252,BLKSIZE=252,RECFM=F)
//S1  EXEC  PGM=GO,REGION=450K,COND=(0,NE,DFC),PARM='NOSTAE,NOSPIE'
```

```
//* DD STATEMENTS FOR THE SIMULATOR STEP
//STEPLIB DD DSN=P.I4987.DFS.NEWSIM,DISP=SHR
//SYSPRINT  DD  SYSOUT=A
//ETC  DD  SYSOUT=A,DCB=(RECFM=FB,LRECL=132,BLKSIZE=132)
//PINFD DD DUMMY,DCB=(RECFM=F,LRECL=132,BLKSIZE=132)
//PFRNG DD DUMMY,DCB=(RECFM=F,LRECL=132,BLKSIZE=132)
//PINST DD DUMMY,DCB=(RECFM=F,LRECL=132,BLKSIZE=132)
//PLIDUMP  DD  SYSOUT=A
//SNAPDUMP  DD  SYSOUT=A
//SYSUDUMP DD SYSOUT=A
//SYSIN  DD  *
TRACE='0'B, FINAL='0'B;
13,500,1,0,100,1
20000,30000,13000,8000,16000
3,3,1,2,15
6,1,5,5,1
1,1,1,1,1
8,8,10,8,5
9,3,3,3,3
3,9,3,3,9
3,1,1,4,4
4,11,1,4,15
20,5,5,5,5
5,5
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
1,1
0201

//MARC DD DSN=J.I4987.ASM.HYSL,DISP=(OLD,KEEP)
```

```
PROCEDURE NEXP BEGIN
  INTEGER N, I;
  REAL S, E, X;
  FILE INFILE, OUTFILE;
  PROCEDURE EXP (IN(X), OUT(E)) BEGIN
    REAL X, E;
    REAL STREAM R;
    INTEGER STREAM P, F;
    PROCEDURE FACT (IN(I, PREV, N), OUT(F)) BEGIN
      INTEGER I, PREV, N, Y;
      INTEGER STREAM F, T;
      IF I > N
        THEN F := 524288
        ELSE BEGIN
          Y := PREV * I;
          FACT(IN(Y, I+1, N), OUT(T));
          F := CONS(IN(Y,T))
          END
      END;
    FACT(IN(1,1,3), OUT(F));
    P := SCREATE(IN(1,3,1));
    R := REPL(IN(X,P)) ** P / F;
    E := SUM(IN(R))
    END;
  INPUT N  FILE=INFILE FORMAT=I(2);
  S := 0.0;
  I := 1;
  WHILE I<=N DO
    INPUT X  FILE=INFILE FORMAT=(SKIP(1),F(5,3));
    EXP(IN(X), OUT(E));
    S := S + (E + 1./E) / 2.;
    I := I + 1
    END;
  OUTPUT S  FILE=OUTFILE FORMAT=F(10,5)
  END
```

```
PROC NEXP
        0 ID (T=S,R='NIL')(T=S,D=1.1,33.1,6.1,5.1)
        1 CONS (T=S)(T=F,R='INFILE',C=C) (T=F,D=2.1)
        2 READ (T=F)(R='I(2)',T=C,C=C)(T=S,D=3.1,4.1)
        3 SELECT (T=S)(R=1,T=I,C=C)(T=F,D=10.2)
        4 SELECT (T=S)(R=2,T=I,C=C)(T=I,D=9.2)
        5 CONS (T=S)(T=R,R=0.0,C=C)(T=R,D=11.2)
        6 CONS (T=S)(T=I,R=1,C=C)(T=I,D=8.2)
        8 MERGE (T=I)(T=I)(G=F,@=7)(T=I,D=7.1,12.1)
        9 MERGE (T=I)(T=I)(G=F,@=7)(T=I,D=7.2,13.1)
       10 MERGE (T=F)(T=F)(G=F,@=7)(T=F,D=14.1)
       11 MERGE (T=R)(T=R)(G=F,@=7)(T=R,D=15.1,32.1)
        7 <= (T=I)(T=I)(C=4)(T=G,D=8.0,32.1,15.1,14.1,13.1,12.1,11.0,
          10.0,9.0)
       12 ID (T=I,C=T)(T=I,D=20.1,31.1)
       13 ID (T=I,C=T)(T=I,D=9.1)
       14 ID (T=F,C=T)(T=F,D=16.1)
       15 ID (T=R,C=T)(T=R,D=30.1)
       16 READEDIT (T=F)(R='SKIP(1)',C=C,T=C)(T=F,D=17.1)
       17 READ (T=F)(R='F(5,3)',T=C,C=C)(T=S,D=18.1,19.1)
       18 SELECT (T=S)(R=1,T=I,C=C)(T=F,D=10.1)
       19 SELECT (T=S)(R=2,T=I,C=C)(T=R,D=24.1)
       20 CONS (T=I)(T=I,C=C,R=1)(T=I,D=21.3)
       21 APPEND (T=S,C=C,R='NIL')(T=I,R=1,C=C)(T=I)(T=S,D=22.1)
       22 APPEND (T=S)(T=I,R=2,C=C)(T=I,C=C,R=1)(T=S,D=23.2)
       23 APPLY* (R='EXP',T=P,C=C)(T=S)(T=I,C=C,R=2)(T=S,D=)
       24 SEND (T=R)(T=R,D=)
       25 REC (T=R)(T=R,D=26.1,28.1,27.2)
       26 ACK (T=R)(T=I)(T=I)(T=I,D=)
       27 / (T=R,R=1.,C=C)(T=R)(T=R,D=28.2)
       28 + (T=R)(T=R)(T=R,D=29.1)
       29 / (T=R)(T=R,R=2.,C=C)(T=R,D=30.2)
       30 + (T=R)(T=R)(T=R,D=11.1)
       31 + (T=I)(T=I,R=1,C=C)(T=I,D=8.1)
       32 ID (T=R,C=F)(T=R,D=34.3)
       33 CONS (T=S)(T=F,R='OUTFILE',C=C) (T=F,D=34.1)
       34 WRITE (T=F)(R='F(10,5)',T=C,C=C)(T=R)(T=F,D=)
```

```
        17 ID (T=I,C=T)(T=I,D=16.1,20.1)
        18 MERGE (T=I)(T=I)(G=F,0=21)(T=I,D=19.1,21.1)
        19 ID (T=I,C=T)(T=I,D=20.2,23.1)
        20 + (T=I)(T=I)(T=I,D=18.1)
        21 <= (T=I)(T=I)(C=3)(T=G,D=16.0,17.1,18.0,19.1,22.0,24.1,23.0)
        22 MERGE (T=I)(T=I)(G=F,0=21)(T=I,D=21.2,24.1)
        23 MERGE (T=I)(T=I,C=C,R=524288)()(T=I,D=25.1,31.2)
        24 ID (T=I,C=T)(T=I,D=22.1)
        25 ID (T=I)(T=I,D=27.1)
        26 ID (T=R)(T=R,D=28.1)
        27 EOS (T=I)(C=1)(T=G,D=28.0,29.1,30.0)
        28 MERGE (T=R)(T=R)(G=T,0=27)(T=R,D=29.1)
        29 ID (T=R,C=F)(T=R,D=28.2,30.2)
        30 MERGE (T=R,C=C,R=524288)(T=R)()(T=R,D=31.1)
        31 ** (T=R)(T=I)(T=R,D=32.1)
        32 / (T=R)(T=I)(T=R,D=33.1,34.1)
        33 ID (T=R)(T=R,D=37.1)
        34 EOS (T=R)(C=1)(T=G,D=36.0,37.1,37.2,38.1)
PROC EXP
         0 SEND (T=R)(T=R,D=)
         1 REC (T=R)(T=R,D=2.1,35.1,26.1,15.1,14.1,13.1,3.1)
         2 ACK (T=R)(T=I)(T=I)(T=I,D=)
         3 CONS (T=R)(T=I,C=C,R=3)(T=I,D=4.3)
         4 APPEND (T=S,C=C,R='NIL')(T=I,R=1,C=C)(T=I)(T=S,D=5.1)
         5 APPEND (T=S)(T=I,R=2,C=C)(T=I,C=C,R=1)(T=S,D=6.2)
         6 APPLY* (R='FACT',T=P,C=C)(T=S)(T=I,C=C,R=4)(T=S,D=)
         7 SEND (T=I,C=C,R=3)(T=I,D=)
         8 SEND (T=I,C=C,R=1)(T=I,D=)
         9 SEND (T=I,C=C,R=1)(T=I,D=)
        10 REC (T=I)(T=I,D=11.1,32.2,12.1)
        11 SACK1 (T=I)(T=I)(T=I)(T=I,D=)
        12 SACK2 (T=I)(T=I)(T=I)(T=I,D=)
        13 CONS (T=R)(T=I,C=C,R=1)(T=I,D=16.2)
        14 CONS (T=R)(T=I,C=C,R=3)(T=I,D=22.2)
        15 CONS (T=R)(T=I,C=C,R=1)(T=I,D=18.2)
        16 MERGE (T=I)(T=I)(G=F,0=21)(T=I,D=17.1)
```

```
      35 CONS (T=R)(T=R,C=C,R=0)(T=R,D=36.1)
      36 MERGE (T=R)(T=R)(G=T,@=34)(T=R,D=37.2,38.1)
      37 + (T=R,C=F)(T=R,C=F)(T=R,D=36.2)
      38 ID (T=R,C=T)(T=R,D=0.1)
PROC FACT
       0 SEND (T=I)(T=I,D=)
       1 REC (T=I)(T=I,D=4.1,12.1,7.2)
       2 REC (T=I)(T=I,D=5.1,10.1)
       3 REC (T=I)(T=I,D=6.1,11.1,8.1,7.1)
       4 ACK (T=I)(T=I)(T=I)(T=I,D=)
       5 ACK (T=I)(T=I)(T=I)(T=I,D=)
       6 ACK (T=I)(T=I)(T=I)(T=I,D=)
       7 > (T=I)(T=I)(C=0)(T=G,D=8.1,29.1.D,12.1,11.1,10.1)
       8 ID (T=I,C=T)(T=I,D=9.1)
       9 CONS (T=I)(T=I,R=524288,C=C)(T=I,D=27.1)
      10 ID (C=F,T=I)(T=I,D=13.1)
      11 ID (C=F,T=I)(T=I,D=13.2,14.1)
      12 ID (C=F,T=I)(T=I,D=15.1,19.1)
      13 * (T=I)(T=I)(T=I,D=21.1,25.1)
      14 + (T=I)(T=I,R=1,C=C)(T=I,D=20.1)
      15 CONS (T=I)(T=I,C=C,R=3)(T=I,D=16.3)
      16 APPEND (T=S,C=C,R='NIL')(T=I,R=1,C=C)(T=I)(T=S,D=17.1)
      17 APPEND (T=S)(T=I,R=2,C=C)(T=I,C=C,R=1)(T=S,D=18.2)
      18 APPLY* (R='FACT',T=P,C=C)(T=S)(T=I,C=C,R=4)(T=S,D=)
      19 SEND (T=I)(T=I,D=)
      20 SEND (T=I)(T=I,D=)
      21 SEND (T=I)(T=I,D=)
      22 REC (T=I)(T=I,D=23.1,25.2,24.1)
      23 SACK1 (T=I)(T=I)(T=I)(T=I,D=)
      24 SACK2 (T=I)(T=I)(T=I)(T=I,D=)
      25 MERGE (T=I)(T=I)(G=T,@=26)(T=I,D=26.1,27.2)
      26 EOS (T=I)(C=1)(T=G,D=25.0)
      27 MERGE (T=I)(T=I)()(T=I,D=28.1,0.1)
      28 EOS (T=I)(C=1)(T=G,D=29.0,29.2)
      29 MERGE (T=G)(C=F,T=G)(G=T,@=28)(T=G,D=27.0,29.2.D)
```

```
//E327NEXP  JOB  I4987,CPLTJCL,MSGLEVEL=(1,1)
//DFC  EXEC  PGM=DFC,REGION=356K,TIME=(,10),PARM=('ISA(-14K)/',
//    'CODE')
//STEPLIB  DD DSN=A.I4987.DFC.NEWCOM2,DISP=SHR
//SYSPRINT  DD  SYSOUT=A
//CODEFIL DD DSN=J.I4987.DFC.CODE1,UNIT=DISK,
//  DISP=(NEW,CATLG,DELETE),
//  DCB=(LRECL=80,BLKSIZE=800,RECFM=FB,BUFNO=1),
//  SPACE=(TRK,(10,1),RLSE)
//LASTFIL DD DSN=J.I4987.CODE.NEXP,UNIT=DISK,
//    DCB=(LRECL=80,BLKSIZE=800,RECFM=FB,BUFNO=1),
//    DISP=(NEW,CATLG,DELETE),SPACE=(TRK,(10,2),RLSE,CONTIG)
//SYSTEST  DD  SYSOUT=A
//SYSDUMP  DD SYSOUT=A
//PLIDUMP  DD  SYSOUT=A,DCB=BLKSIZE=133
//SYSIN  DD *
 PROCEDURE NEXP BEGIN
   INTEGER N, I;
   REAL S, E, X;
   FILE INFILE, OUTFILE;
   PROCEDURE EXP (IN(X), OUT(E)) BEGIN
     REAL X, E;
     REAL STREAM R;
     INTEGER STREAM P, F;
     PROCEDURE FACT (IN(I, PREV, N), OUT(F)) BEGIN
       INTEGER I, PREV, N, Y;
       INTEGER STREAM F, T;
       IF I > N
         THEN F := 524288
         ELSE BEGIN
           Y := PREV * I;
           FACT(IN(Y, I+1, N), OUT(T));
           F := CONS(IN(Y,T))
           END
       END;
     FACT(IN(1,1,3), OUT(F));
     P := SCREATE(IN(1,3,1));
     R := REPL(IN(X,P)) ** P / F;
     E := SUM(IN(R))
     END;
   INPUT N  FILE=INFILE FORMAT=I(2);
   S := 0.0;
   I := 1;
   WHILE I<=N DO
     INPUT X  FILE=INFILE FORMAT=(SKIP(1),F(5,3));
     EXP(IN(X), OUT(E));
     S := S + (E + 1./E) / 2.;
     I := I + 1
     END;
   OUTPUT S  FILE=OUTFILE FORMAT=F(10,5)
   END
```

```
//STEP2 EXEC PGM=IEBPTPCH,COND=(0,EQ,DFC)
//SYSPRINT DD SYSOUT=A
//SYSUT1 DD DSN=J.I4987.DFC.CODE1,DISP=(OLD,DELETE)
//SYSUT2 DD SYSOUT=A
//SYSIN DD *
 PRINT MAXFLDS=1
 TITLE ITEM=('NEXP')
 RECORD FIELD=(80)
/*
//DEL DD DSN=J.I4987.CODE.NEXP,DISP=(OLD,DELETE)
//XS1 EXEC PGM=IEFBR14,COND=(0,NE,DFC)
//DD1 DD DSN=J.I4987.DFC.CODE1,DISP=(OLD,DELETE)
//ASM EXEC PGM=GO,REGION=100K,TIME=(,10),COND=(0,NE,DFC)
//* DD STATEMENTS FOR ASSEMBLER STEP
//STEPLIB DD DSN=P.I4119.ASMLOAD,DISP=SHR
//SYSIN DD DSN=J.I4987.CODE.NEXP,DISP=(OLD,DELETE)
//SYSPRINT DD SYSOUT=A
//PLIDUMP DD SYSOUT=A,DCB=BLKSIZE=133
//SYSUDUMP DD SYSOUT=A
//MARC DD DSN=J.I4987.ASM.NEXP,UNIT=DISK,DISP=(NEW,CATLG,DELETE),
//  SPACE=(TRK,(10,1),RLSE),
// DCB=(DSORG=DA,LRECL=252,BLKSIZE=252,RECFM=F)
//S1  EXEC  PGM=GO,REGION=450K,COND=(0,NE,DFC),PARM='NOSTAE,NOSPIE'
//* DD STATEMENTS FOR THE SIMULATOR STEP
//STEPLIB DD DSN=P.I4987.DFS.NEWSIM,DISP=SHR
//SYSPRINT  DD  SYSOUT=A
```

```
//ETC  DD  SYSOUT=A,DCB=(RECFM=FB,LRECL=132,BLKSIZE=132)
//PINFO DD DUMMY,DCB=(RECFM=F,LRECL=132,BLKSIZE=132)
//PFRNG DD DUMMY,DCB=(RECFM=F,LRECL=132,BLKSIZE=132)
//PINST DD DUMMY,DCB=(RECFM=F,LRECL=132,BLKSIZE=132)
//PLIDUMP  DD  SYSOUT=A
//SNAPDUMP  DD  SYSOUT=A
//SYSUDUMP DD SYSOUT=A
//SYSIN  DD  *
TRACE='1'B, FINAL='0'B;
13,500,1,0,100,1
20000,30000,13000,8000,16000
3,3,1,2,15
6,1,5,5,1
1,1,1,1,1
8,8,10,8,5
3,3,3,3,3
3,1,3,3,9
3,1,1,4,4
4,11,1,4,15
20,5,5,5,5
5,5
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
1,1
01
01250

//MARC DD DSN=J.I4987.ASM.NEXP,DISP=(OLD,KEEP)
```

```
PROCEDURE TRIG BEGIN
  REAL STREAM A, B;
  REAL Z;
  FILE INFILE, OUTFILE;
  PROCEDURE SINK(IN(X,I), OUT(V)) BEGIN
    INTEGER I;
    REAL STREAM X;
    REAL V, W, Y;
    IF EMPTY(IN(X))
      THEN V := 0.0
      ELSE BEGIN
        W := (FIRST(IN(X)) / I) ** 2;
        SINK(IN(REST(IN(X)), I+1), OUT(Y));
        V := W + Y
        END
    END;
  STREAM INPUT A  FILE=INFILE FORMAT=F(7,1);
  B := (SIN(IN(A)) + COS(IN(A))) / REPL(IN(2,A));
  SINK(IN(B,1), OUT(Z));
  OUTPUT Z  FILE=OUTFILE FORMAT=F(10,5)
  END
```

```
PROC TRIG
        0 ID (T=S,R='NIL')(T=S,D=1.1,26.1,18.1,12.1)
        1 CONS (T=S)(T=F,R='INFILE',C=C) (T=F,D=2.1)
        2 CONS (T=F)(T=F,C=C,R='INFILE')(T=F,D=3.1)
        3 READ (T=F)(R='F(7,1)',T=C,C=C)(T=S,D=4.1,5.1)
        4 SELECT (T=S)(R=1,T=I,C=C)(T=F,D=6.1)
        5 SELECT (T=S)(R=2,T=I,C=C)(T=R,D=7.1,11.1,9.1,8.1)
        6 ID (T=F,C=F)(T=F,D=2.1)
        7 EOS (T=R)(T=G,D=6.1)
        8 SIN (T=R)(T=R,D=10.1)
        9 COS (T=R)(T=R,D=10.2)
       10 + (T=R)(T=R)(T=R,D=17.1)
       11 ID (T=R)(T=R,D=13.1)
       12 CONS (T=S)(T=I,C=C,R=2)(T=I,D=14.1)
       13 EOS (T=R)(C=1)(T=G,D=14.0,15.1,16.0)
       14 MERGE (T=I)(T=I)(G=T,@=13)(T=I,D=15.1)
       15 ID (T=I,C=F)(T=I,D=14.2,16.2)
       16 MERGE (T=I,C=C,R=524288)(T=I)()(T=I,D=17.2)
       17 / (T=R)(T=I)(T=R,D=23.1)
       18 CONS (T=S)(T=I,C=C,R=2)(T=I,D=19.3)
       19 APPEND (T=S,C=C,R='NIL')(T=I,R=1,C=C)(T=I)(T=S,D=20.1)
       20 APPEND (T=S)(T=I,R=2,C=C)(T=I,C=C,R=1)(T=S,D=21.2)
       21 APPLY* (R='SINK',T=P,C=C)(T=S)(T=I,C=C,R=3)(T=S,D=)
       22 SEND (T=I,C=C,R=1)(T=I,D=)
       23 SEND (T=R)(T=R,D=)
       24 REC (T=R)(T=R,D=25.1,27.3)
       25 ACK (T=R)(T=I)(T=I)(T=I,D=)
       26 CONS (T=S)(T=F,R='OUTFILE',C=C) (T=F,D=27.1)
       27 WRITE (T=F)(R='F(10,5)',T=C,C=C)(T=R)(T=F,D=)
```

```
PROC SINK
        0 SEND (T=R)(T=R,D=)
        1 REC (T=I)(T=I,D=3.1,15.1,10.1)
        2 REC (T=R)(T=R,D=4.1,13.1,12.1,6.1,5.1)
        3 ACK (T=I)(T=I)(T=I)(T=I,D=)
        4 SACK1 (T=R)(T=I)(T=I)(T=1,D=)
        5 SACK2 (T=R)(T=I)(T=I)(T=I,D=)
        6 ID (T=R)(T=R,D=7.1,8.1)
        7 EOS (T=R)(C=1)(T=G,D=8.1)
        8 EOS (T=R,C=T,G=T,@=7)(T=B,D=9.1)
        9 ID (T=B)(C=0)(T=G,D=10.1,34.0,15.1,14.1.D)
       10 ID (T=I,C=T)(T=I,D=11.1)
       11 CONS (T=I)(T=R,R=0.0,C=C)(T=R,D=34.1)
       12 ID (C=F,T=R)(T=R,D=16.1,22.1)
       13 EOS (T=R)(C=1)(T=G,D=14.0,14.2)
       14 MERGE (T=G)(T=G,C=F)(G=T,@=13)(T=G,D=12.1,14.2.D)
       15 ID (C=F,T=I)(T=I,D=19.2,25.1,21.1)
       16 ID (T=R)(T=R,D=17.1,18.1)
       17 EOS (T=R)(C=1)(T=G,D=18.1)
       18 ID (T=R,C=T,G=T,@=17)(T=R,D=19.1)
       19 / (T=R)(T=I)(T=R,D=20.1)
       20 ** (T=R)(T=I,R=2,C=C)(T=R,D=33.1)
       21 + (T=I)(T=I,R=1,C=C)(T=I,D=29.1)
       22 ID (T=R)(T=R,D=23.1,24.1)
       23 EOS (T=R)(C=1)(T=G,D=24.1)
       24 ID (T=R,C=F,G=T,@=23)(T=R,D=30.1)
       25 CONS (T=I)(T=I,C=C,R=2)(T=I,D=26.3)
       26 APPEND (T=S,C=C,R='NIL')(T=I,R=1,C=C)(T=I)(T=S,D=27.1)
       27 APPEND (T=S)(T=I,R=2,C=C)(T=I,C=C,R=1)(T=S,D=28.2)
       28 APPLY* (R='SINK',T=P,C=C)(T=S)(T=I,C=C,R=3)(T=S,D=)
       29 SEND (T=I)(T=I,D=)
       30 SEND (T=R)(T=R,D=)
       31 REC (T=R)(T=R,D=32.1,33.2)
       32 ACK (T=R)(T=I)(T=I)(T=I,D=)
       33 + (T=R)(T=R)(T=R,D=34.2)
       34 MERGE (T=R)(T=R)()(T=R,D=0.1)
```

```
//E327TRIG  JOB  I4987,CPLTJCL,MSGLEVEL=(1,1)
//DFC  EXEC  PGM=DFC,REGION=356K,TIME=(,10),PARM=('ISA(-14K)/',
//    'CODE')
//STEPLIB  DD DSN=A.I4987.DFC.NEWCOM2,DISP=SHR
//SYSPRINT  DD  SYSOUT=A
//CODEFIL DD DSN=J.I4987.DFC.CODE1,UNIT=DISK,
//  DISP=(NEW,CATLG,DELETE),
//  DCB=(LRECL=80,BLKSIZE=800,RECFM=FB,BUFNO=1),
//  SPACE=(TRK,(10,1),RLSE)
//LASTFIL DD DSN=J.I4987.CODE.TRIG,UNIT=DISK,
//    DCB=(LRECL=80,BLKSIZE=800,RECFM=FB,BUFNO=1),
//    DISP=(NEW,CATLG,DELETE),SPACE=(TRK,(10,2),RLSE,CONTIG)
//SYSTEST  DD  SYSOUT=A
//SYSDUMP  DD SYSOUT=A
//PLIDUMP  DD  SYSOUT=A,DCB=BLKSIZE=133
//SYSIN  DD *
 PROCEDURE TRIG BEGIN
   REAL STREAM A, B;
   REAL Z;
   FILE INFILE, OUTFILE;
   PROCEDURE SINK(IN(X,I), OUT(V)) BEGIN
     INTEGER I;
     REAL STREAM X;
     REAL V, W, Y;
     IF EMPTY(IN(X))
       THEN V := 0.0
       ELSE BEGIN
         W := (FIRST(IN(X)) / I) ** 2;
         SINK(IN(REST(IN(X)), I+1), OUT(Y));
         V := W + Y
         END
     END;
   STREAM INPUT A  FILE=INFILE FORMAT=F(7,1);
   B := (SIN(IN(A)) + COS(IN(A))) / REPL(IN(2,A));
   SINK(IN(B,1), OUT(Z));
   OUTPUT Z  FILE=OUTFILE FORMAT=F(10,5)
   END

//STEP2 EXEC PGM=IEBPTPCH,COND=(0,EQ,DFC)
//SYSPRINT DD SYSOUT=A
//SYSUT1 DD DSN=J.I4987.DFC.CODE1,DISP=(OLD,DELETE)
//SYSUT2 DD SYSOUT=A
//SYSIN DD *
 PRINT MAXFLDS=1
 TITLE ITEM=('TRIG')
 RECORD FIELD=(80)
/*
//DEL  DD DSN=J.I4987.CODE.TRIG,DISP=(OLD,DELETE)
//XS1 EXEC PGM=IEFBR14,COND=(0,NE,DFC)
//DD1 DD DSN=J.I4987.DFC.CODE1,DISP=(OLD,DELETE)
```

```
//ASM EXEC PGM=GO,REGION=100K,TIME=(,10),COND=(0,NE,DFC)
//* DD STATEMENTS FOR ASSEMBLER STEP
//STEPLIB DD DSN=P.I4119.ASMLOAD,DISP=SHR
//SYSIN DD DSN=J.I4987.CODE.TRIG,DISP=(OLD,DELETE)
//SYSPRINT DD SYSOUT=A
//PLIDUMP DD SYSOUT=A,DCB=BLKSIZE=133
//SYSUDUMP DD SYSOUT=A
//MARC DD DSN=J.I4987.ASM.TRIG,UNIT=DISK,DISP=(NEW,CATLG,DELETE),
//  SPACE=(TRK,(10,1),RLSE),
// DCB=(DSORG=DA,LRECL=252,BLKSIZE=252,RECFM=F)
//S1  EXEC  PGM=GO,REGION=450K,COND=(0,NE,DFC),PARM='NOSTAE,NOSPIE'
//* DD STATEMENTS FOR THE SIMULATOR STEP
//STEPLIB DD DSN=P.I4987.DFS.NEWSIM,DISP=SHR
//SYSPRINT  DD  SYSOUT=A
//ETC  DD  SYSOUT=A,DCB=(RECFM=FB,LRECL=132,BLKSIZE=132)
//PINFO DD DUMMY,DCB=(RECFM=F,LRECL=132,BLKSIZE=132)
//PFRNG DD DUMMY,DCB=(RECFM=F,LRECL=132,BLKSIZE=132)
//PINST DD DUMMY,DCB=(RECFM=F,LRECL=132,BLKSIZE=132)
//PLIDUMP  DD  SYSOUT=A
//SNAPDUMP  DD  SYSOUT=A
//SYSUDUMP DD SYSOUT=A
```

```
//SYSIN  DD  *
TRACE='1'B, FINAL='0'B;
13,500,1,0,100,1
20000,30000,13000,8000,16000
3,3,1,2,15
6,1,5,5,1
1,1,1,1,1
8,8,10,8,5
9,3,3,3,3
3,9,3,3,9
3,1,1,4,4
4,11,1,4,15
20,5,5,5,5
5,5
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
1,1
00000100000325242880

//MARC DD DSN=J.I4987.ASM.TRIG,DISP=(OLD,KEEP)
```

```
PROCEDURE SSUM BEGIN
  INTEGER STREAM A, B, X, Y;
  INTEGER N, Z;
  FILE INFILE, OUTFILE;
  PROCEDURE P(IN(A), OUT(B)) BEGIN
    INTEGER S;
    INTEGER STREAM A, B, C, D;
    IF EMPTY(IN(A))
      THEN B := 524288
      ELSE BEGIN
        C := SCREATE(IN(1, FIRST(IN(A)), 1));
        S := SUM(IN(C * C));
        P(IN(REST(IN(A))), OUT(D));
        B := CONS(IN(S, D))
        END
    END;
  INPUT N  FILE=INFILE FORMAT=I(2);
  STREAM INPUT A  FILE=INFILE FORMAT=I(6);
  X := SCREATE(IN(1, N, 1));
  P(IN(A), OUT(B));
  Y := X * B;
  Z := SUM(IN(Y));
  OUTPUT Z  FILE=OUTFILE FORMAT=I(10)
  END
```

```
PROC SSUM
     0 ID (T=S,R='NIL')(T=S,D=1.1,38.1,34.1,23.1,13.1,11.1)
     1 CONS (T=S)(T=F,R='INFILE',C=C) (T=F,D=2.1)
     2 READ (T=F)(R='I(2)',T=C,C=C)(T=S,D=3.1,4.1)
     3 SELECT (T=S)(R=1,T=I,C=C)(T=F,D=5.1)
     4 SELECT (T=S)(R=2,T=I,C=C)(T=I,D=12.1)
     5 CONS (T=F)(T=F,C=C,R='INFILE')(T=F,D=6.1)
     6 READ (T=F)(R='I(6)',T=C,C=C)(T=S,D=7.1,8.1)
     7 SELECT (T=S)(R=1,T=I,C=C)(T=F,D=9.1)
     8 SELECT (T=S)(R=2,T=I,C=C)(T=I,D=10.1,27.1)
     9 ID (T=F,C=F)(T=F,D=5.1)
    10 EOS (T=I)(T=G,D=9.1)
    11 CONS (T=S)(T=I,C=C,R=1)(T=I,D=14.2)
    12 ID (T=I)(T=I,D=20.2)
    13 CONS (T=S)(T=I,C=C,R=1)(T=I,D=16.2)
    14 MERGE (T=I)(T=I)(G=F,@=19)(T=I,D=15.1)
    15 ID (T=I,C=T)(T=I,D=14.1,18.1)
    16 MERGE (T=I)(T=I)(G=F,@=19)(T=I,D=17.1,19.1)
    17 ID (T=I,C=T)(T=I,D=18.2,21.1)
    18 + (T=I)(T=I)(T=I,D=16.1)
    19 <= (T=I)(T=I)(C=3)(T=G,D=14.0,15.1,16.0,17.1,20.0,22.1,21.0)
    20 MERGE (T=I)(T=I)(G=F,@=19)(T=I,D=19.2,22.1)
    21 MERGE (T=I)(T=I,C=C,R=524288)()(T=I,D=31.1)
    22 ID (T=I,C=T)(T=I,D=20.1)
    23 CONS (T=S)(T=I,C=C,R=1)(T=I,D=24.3)
    24 APPEND (T=S,C=C,R='NIL')(T=I,R=1,C=C)(T=I)(T=S,D=25.1)
    25 APPEND (T=S)(T=I,R=2,C=C)(T=I,C=C,R=1)(T=S,D=26.2)
    26 APPLY* (R='P',T=P,C=C)(T=S)(T=I,C=C,R=2)(T=S,D=)
    27 SEND (T=I)(T=I,D=)
    28 REC (T=I)(T=I,D=29.1,31.2,30.1)
    29 SACK1 (T=I)(T=I)(T=I)(T=I,D=)
    30 SACK2 (T=I)(T=I)(T=I)(T=I,D=)
    31 * (T=I)(T=I)(T=I,D=32.1,33.1)
```

```
      32 ID (T=I)(T=I,D=36.1)
      33 EOS (T=I)(C=1)(T=G,D=35.0,36.1,36.2,37.1)
      34 CONS (T=S)(T=I,C=C,R=0)(T=I,D=35.1)
      35 MERGE (T=I)(T=I)(G=T,@=33)(T=I,D=36.2,37.1)
      36 + (T=I,C=F)(T=I,C=F)(T=I,D=35.2)
      37 ID (T=I,C=T)(T=I,D=39.3)
      38 CONS (T=S)(T=F,R='OUTFILE',C=C) (T=F,D=39.1)
      39 WRITE (T=F)(R='I(10)',T=C,C=C)(T=I)(T=F,D=)
PROC P
       0 SEND (T=I)(T=I,D=)
       1 REC (T=I)(T=I,D=2.1,12.1,11.1,5.1,3.1)
       2 SACK1 (T=I)(T=I)(T=I)(T=I,D=)
       3 SACK2 (T=I)(T=I)(T=I)(T=I,D=)
       4 ID (T=S,R='NIL')(T=S,D=9.1,14.1)
       5 ID (T=I)(T=I,D=6.1,7.1)
       6 EOS (T=I)(C=1)(T=G,D=7.1)
       7 EOS (T=I,C=T,G=T,@=6)(T=B,D=8.1)
       8 ID (T=B)(C=0)(T=G,D=9.1,52.1,D,14.1,13.1,D)
       9 ID (T=S,C=T)(T=S,D=10.1)
      10 CONS (T=S)(T=I,R=524288,C=C)(T=I,D=50.1)
      11 ID (C=F,T=I)(T=I,D=16.1,37.1)
      12 EOS (T=I)(C=1)(T=G,D=13.0,13.2)
      13 MERGE (T=G)(T=G,C=F)(G=T,@=12)(T=G,D=11.1,13.2,D)
      14 ID (T=S,C=F)(T=S,D=15.1,40.1,33.1,20.1)
      15 CONS (T=S)(T=I,C=C,R=1)(T=I,D=21.2)
      16 ID (T=I)(T=I,D=17.1,18.1)
      17 EOS (T=I)(C=1)(T=G,D=18.1)
      18 ID (T=I,C=T,G=T,@=17)(T=I,D=19.1)
      19 ID (T=I)(T=I,D=27.2)
      20 CONS (T=S)(T=I,C=C,R=1)(T=I,D=23.2)
      21 MERGE (T=I)(T=I)(G=F,@=26)(T=I,D=22.1)
```

```
22 ID (T=I,C=T)(T=I,D=21.1,25.1)
23 MERGE (T=I)(T=I)(G=F,@=26)(T=I,D=24.1,26.1)
24 ID (T=I,C=T)(T=I,D=25.2,28.1)
25 + (T=I)(T=I)(T=I,D=23.1)
26 <= (T=I)(T=I)(C=3)(T=G,D=21.0,22.1,23.0,24.1,27.0,29.1,28.0)
27 MERGE (T=I)(T=I)(G=F,@=26)(T=I,D=26.2,29.1)
28 MERGE (T=I)(T=I,C=C,R=524288)()(T=I,D=30.1,30.2)
29 ID (T=I,C=T)(T=I,D=27.1)
30 * (T=I)(T=I)(T=I,D=31.1,32.1)
31 ID (T=I)(T=I,D=35.1)
32 EOS (T=I)(C=1)(T=G,D=34.0,35.1,35.2,36.1)
33 CONS (T=S)(T=I,C=C,R=0)(T=I,D=34.1)
34 MERGE (T=I)(T=I)(G=T,@=32)(T=I,D=35.2,36.1)
35 + (T=I,C=F)(T=I,C=F)(T=I,D=34.2)
36 ID (T=I,C=T)(T=I,D=48.1)
37 ID (T=I)(T=I,D=38.1,39.1)
38 EOS (T=I)(C=1)(T=G,D=39.1)
39 ID (T=I,C=F,G=T,@=38)(T=I,D=44.1)
40 CONS (T=S)(T=I,C=C,R=1)(T=I,D=41.3)
41 APPEND (T=S,C=C,R='NIL')(T=I,R=1,C=C)(T=I)(T=S,D=42.1)
42 APPEND (T=S)(T=I,R=2,C=C)(T=I,C=C,R=1)(T=S,D=43.2)
43 APPLY* (R='P',T=P,C=C)(T=S)(T=I,C=C,R=2)(T=S,D=)
44 SEND (T=I)(T=I,D=)
45 REC (T=I)(T=I,D=46.1,48.2,47.1)
46 SACK1 (T=I)(T=I)(T=I)(T=I,D=)
47 SACK2 (T=I)(T=I)(T=I)(T=I,D=)
48 MERGE (T=I)(T=I)(G=T,@=49)(T=I,D=49.1,50.2)
49 EOS (T=I)(C=1)(T=G,D=48.0)
50 MERGE (T=I)(T=I)()(T=I,D=51.1,0.1)
51 EOS (T=I)(C=1)(T=G,D=52.0,52.2)
52 MERGE (T=G)(C=F,T=G)(G=T,@=51)(T=G,D=50.0,52.2,D)
```

```
//E327SSUM  JOB  I4987,CPLTJCL,MSGLEVEL=(1,1)
/*JOBPARM  LINES=10
//DFC  EXEC  PGM=DFC,REGION=356K,TIME=(,10),PARM=('ISA(-14K)/',
//   'CODE')
//STEPLIB  DD DSN=A.I4987.DFC.NEWCOM2,DISP=SHR
//SYSPRINT  DD  SYSOUT=A
//CODEFIL DD DSN=J.I4987.DFC.CODE1,UNIT=DISK,
//  DISP=(NEW,CATLG,DELETE),
//  DCB=(LRECL=80,BLKSIZE=800,RECFM=FB,BUFNO=1),
//  SPACE=(TRK,(10,1),RLSE)
//LASTFIL DD DSN=J.I4987.CODE.SSUM,UNIT=DISK,
//   DCB=(LRECL=80,BLKSIZE=800,RECFM=FB,BUFNO=1),
//   DISP=(NEW,CATLG,DELETE),SPACE=(TRK,(10,2),RLSE,CONTIG)
//SYSTEST  DD  SYSOUT=A
//SYSDUMP  DD SYSOUT=A
//PLIDUMP  DD  SYSOUT=A,DCB=BLKSIZE=133
//SYSIN  DD *
 PROCEDURE SSUM BEGIN
   INTEGER STREAM A, B, X, Y;
   INTEGER N, Z;
   FILE INFILE, OUTFILE;
   PROCEDURE P(IN(A), OUT(B)) BEGIN
     INTEGER S;
     INTEGER STREAM A, B, C, D;
     IF EMPTY(IN(A))
       THEN B := 524288
       ELSE BEGIN
         C := SCREATE(IN(1, FIRST(IN(A)), 1));
         S := SUM(IN(C * C));
         P(IN(REST(IN(A))), OUT(D));
         B := CONS(IN(S, D))
         END
     END;
   INPUT N  FILE=INFILE FORMAT=I(2);
   STREAM INPUT A  FILE=INFILE FORMAT=I(6);
   X := SCREATE(IN(1, N, 1));
   P(IN(A), OUT(B));
   Y := X * B;
   Z := SUM(IN(Y));
   OUTPUT Z  FILE=OUTFILE FORMAT=I(10)
   END

//STEP2 EXEC PGM=IEBPTPCH,COND=(0,EQ,DFC)
//SYSPRINT DD SYSOUT=A
//SYSUT1 DD DSN=J.I4987.DFC.CODE1,DISP=(OLD,DELETE)
//SYSUT2 DD SYSOUT=A
//SYSIN DD *
 PRINT MAXFLDS=1
 TITLE ITEM=('SSUM')
 RECORD FIELD=(80)
/*
```

```
//DEL DD DSN=J.I4987.CODE.SSUM,DISP=(OLD,DELETE)
//XS1 EXEC PGM=IEFBR14,COND=(0,NE,DFC)
//DD1 DD DSN=J.I4987.DFC.CODE1,DISP=(OLD,DELETE)
//ASM EXEC PGM=GO,REGION=100K,TIME=(,10),COND=(0,NE,DFC)
//* DD STATEMENTS FOR ASSEMBLER STEP
//STEPLIB DD DSN=P.I4119.ASMLOAD,DISP=SHR
//SYSIN DD DSN=J.I4987.CODE.SSUM,DISP=(OLD,DELETE)
//SYSPRINT DD SYSOUT=A
//PLIDUMP DD SYSOUT=A,DCB=BLKSIZE=133
//SYSUDUMP DD SYSOUT=A
//MARC DD DSN=J.I4987.ASM.SSUM,UNIT=DISK,DISP=(NEW,CATLG,DELETE),
//  SPACE=(TRK,(10,1),RLSE),
// DCB=(DSORG=DA,LRECL=252,BLKSIZE=252,RECFM=F)
//S1  EXEC  PGM=GO,REGION=450K,COND=(0,NE,DFC),PARM='NOSTAE,NOSPIE'
//* DD STATEMENTS FOR THE SIMULATOR STEP
//STEPLIB DD DSN=P.I4987.DFS.NEWSIM,DISP=SHR
//SYSPRINT  DD  SYSOUT=A
//ETC  DD  SYSOUT=A,DCB=(RECFM=FB,LRECL=132,BLKSIZE=132)
//PINFO DD DUMMY,DCB=(RECFM=F,LRECL=132,BLKSIZE=132)
//PFRNG DD DUMMY,DCB=(RECFM=F,LRECL=132,BLKSIZE=132)
//PINST DD DUMMY,DCB=(RECFM=F,LRECL=132,BLKSIZE=132)
//PLIDUMP  DD  SYSOUT=A
//SNAPDUMP  DD  SYSOUT=A
//SYSUDUMP DD SYSOUT=A
```

```
//SYSIN  DD  *
TRACE='1'B, FINAL='0'B;
13,500,1,0,100,1
20000,30000,13000,8000,16000
3,3,1,2,15
6,1,5,5,1
1,1,1,1,1
8,8,10,8,5
9,3,3,3,3
3,9,3,3,9
3,1,1,4,4
4,11,1,4,15
20,5,5,5,5
5,5
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
1,1
02000002000002524288

//MARC DD DSN=J.I4987.ASM.SSUM.DISP=(OLD,KEEP)
```

```
PROCEDURE MULT BEGIN
  REAL ARRAY AXX(1:10), AX2(1:10), AX3(1:10);
  REAL Z;
  INTEGER NPL;
  INTEGER STREAM S;
  REAL STREAM XX, X2, X3, R;
  FILE INFILE, OUTFILE;
  PROCEDURE P(IN(K, NPL, XX, X2), OUT(X3)) BEGIN
    INTEGER K, NPL, M;
    REAL X;
    REAL STREAM XX, X2, X3, Y;
    IF K > NPL
      THEN X3 := 524288.
      ELSE BEGIN
        M := NPL - K + 1;
        X := SUM(IN(SUBSTM(IN(XX,0,M)) *
                    SUBSTM(IN(X2,K-1,M)))) / 2;
        P(IN(K+1, NPL, XX, X2), OUT(Y));
        X3 := CONS(IN(X, Y))
        END
    END;
  PROCEDURE Q(IN(K, NPL, XX, X2), OUT(R)) BEGIN
    INTEGER K, NPL;
    REAL STREAM XX, X2, R, Y;
    REAL X;
    IF K > NPL
      THEN R := 524288.
      ELSE BEGIN
        X := SUM(IN(SUBSTM(IN(XX,0,K-1)) *
                    SUBSTM(IN(X2,0,K-1)))) / 2;
        Q(IN(K+1, NPL, XX, X2), OUT(Y));
        R := CONS(IN(X, Y))
        END
    END;
  INPUT NPL  FILE=INFILE FORMAT=I(2);
  STREAM INPUT XX  FILE=INFILE FORMAT=F(7,1);
  S := SCREATE(IN(1,NPL,1));
  X2 := S;
  P(IN(1, NPL, XX, X2), OUT(X3));
  BUF XX, AXX;
  BUF X2, AX2;
  BUF X3, AX3;
  XX := UNBUF(IN(AXX));
  X2 := UNBUF(IN(AX2));
  X3 := UNBUF(IN(AX3));
  Q(IN(1, NPL, XX, X2), OUT(R));
  Z := SUM(IN(X3 + R));
  OUTPUT Z  FILE=OUTFILE FORMAT=F(10,5)
  END
```

```
PROC MULT
        0 ID (T=S,R='NIL')(T=S,D=1.1,98.1,94.1,80.1,24.1,13.1,11.1)
        1 CONS (T=S)(T=F,R='INFILE',C=C) (T=F,D=2.1)
        2 READ (T=F)(R='I(2)',T=C,C=C)(T=S,D=3.1,4.1)
        3 SELECT (T=S)(R=1,T=I,C=C)(T=F,D=5.1)
        4 SELECT (T=S)(R=2,T=I,C=C)(T=I,D=12.1,86.1,30.1)
        5 CONS (T=F)(T=F,C=C,R='INFILE')(T=F,D=6.1)
        6 READ (T=F)(R='F(7,1)',T=C,C=C)(T=S,D=7.1,8.1)
        7 SELECT (T=S)(R=1,T=I,C=C)(T=F,D=9.1)
        8 SELECT (T=S)(R=2,T=I,C=C)(T=R,D=10.1,35.1,29.1)
        9 ID (T=F,C=F)(T=F,D=5.1)
       10 EOS (T=R)(T=G,D=9.1)
       11 CONS (T=S)(T=I,C=C,R=1)(T=I,D=14.2)
       12 ID (T=I)(T=I,D=20.2)
       13 CONS (T=S)(T=I,C=C,R=1)(T=I,D=16.2)
       14 MERGE (T=I)(T=I)(G=F,@=19)(T=I,D=15.1)
       15 ID (T=I,C=T)(T=I,D=14.1,18.1)
       16 MERGE (T=I)(T=I)(G=F,@=19)(T=I,D=17.1,19.1)
       17 ID (T=I,C=T)(T=I,D=18.2,21.1)
       18 + (T=I)(T=I)(T=I,D=16.1)
       19 <= (T=I)(T=I)(C=3)(T=G,D=14.0,15.1,16.0,17.1,20.0,22.1,21.0)
       20 MERGE (T=I)(T=I)(G=F,@=19)(T=I,D=19.2,22.1)
       21 MERGE (T=I)(T=I,C=C,R=524288)()(T=I,D=23.1)
       22 ID (T=I,C=T)(T=I,D=20.1)
       23 ID (T=I)(T=R,D=28.1,43.1)
       24 CONS (T=S)(T=I,C=C,R=4)(T=I,D=25.3)
       25 APPEND (T=S,C=C,R='NIL')(T=I,R=1,C=C)(T=I)(T=S,D=26.1)
       26 APPEND (T=S)(T=I,R=2,C=C)(T=I,C=C,R=1)(T=S,D=27.2)
       27 APPLY* (R='P',T=P,C=C)(T=S)(T=I,C=C,R=5)(T=S,D=)
       28 SEND (T=R)(T=R,D=)
       29 SEND (T=R)(T=R,D=)
       30 SEND (T=I)(T=I,D=)
       31 SEND (T=I,C=C,R=1)(T=I,D=)
       32 REC (T=R)(T=R,D=33.1,51.1,34.1)
       33 SACK1 (T=R)(T=I)(T=I)(T=I,D=)
       34 SACK2 (T=R)(T=I)(T=I)(T=I,D=)
       35 ID (T=R)(T=R,D=36.1,41.3)
```

```
36 EOS (T=R)(C=2)(T=G,D=37.0,38.1,40.0,41.1,41.3,42.1)
37 MERGE (T=I,C=C,R=1)(T=I)(G=T,@=36)(T=I,D=38.1)
38 ID (T=I,C=F)(T=I,D=39.1,41.2)
39 + (T=I)(T=I,C=C,R=1)(T=I,D=37.2)
40 MERGE (T=S,C=C,R='NIL')(T=S)(G=T,@=36)(T=S,D=41.1,42.1)
41 APPEND (T=S,C=F)(T=I)(T=R,C=F)(T=S,D=40.2)
42 ID (T=S,C=T)(T=S,D=59.1)
43 ID (T=R)(T=R,D=44.1,49.3)
44 EOS (T=R)(C=2)(T=G,D=45.0,46.1,48.0,49.1,49.3,50.1)
45 MERGE (T=I,C=C,R=1)(T=I)(G=T,@=44)(T=I,D=46.1)
46 ID (T=I,C=F)(T=I,D=47.1,49.2)
47 + (T=I)(T=I,C=C,R=1)(T=I,D=45.2)
48 MERGE (T=S,C=C,R='NIL')(T=S)(G=T,@=44)(T=S,D=49.1,50.1)
49 APPEND (T=S,C=F)(T=I)(T=R,C=F)(T=S,D=48.2)
50 ID (T=S,C=T)(T=S,D=66.1)
51 ID (T=R)(T=R,D=52.1,57.3)
52 EOS (T=R)(C=2)(T=G,D=53.0,54.1,56.0,57.1,57.3,58.1)
53 MERGE (T=I,C=C,R=1)(T=I)(G=T,@=52)(T=I,D=54.1)
54 ID (T=I,C=F)(T=I,D=55.1,57.2)
55 + (T=I)(T=I,C=C,R=1)(T=I,D=53.2)
56 MERGE (T=S,C=C,R='NIL')(T=S)(G=T,@=52)(T=S,D=57.1,58.1)
57 APPEND (T=S,C=F)(T=I)(T=R,C=F)(T=S,D=56.2)
58 ID (T=S,C=T)(T=S,D=73.1)
59 ID (T=S)(T=S,D=60.2)
60 MERGE (T=S,C=T)(T=S)(G=F,@=63)(T=S,D=60.1,63.1,64.1)
61 MERGE (T=I,C=T)(T=I,C=C,R=1)(G=F,@=63)(T=I,D=62.1,63.2,64.2)
62 + (T=I)(T=I,C=C,R=1)(T=I,D=61.1)
63 EXISTS (T=S)(T=I)(C=2)(T=G,D=60.0,60.1,61.0,61.1,64.1,64.2,
   65.0)
64 SELECT (T=S,C=T)(T=I,C=T)(T=R,D=65.1)
65 MERGE (T=R)(T=R,C=C,R=524288)()(T=R,D=85.1)
66 ID (T=S)(T=S,D=67.2)
67 MERGE (T=S,C=T)(T=S)(G=F,@=70)(T=S,D=67.1,70.1,71.1)
68 MERGE (T=I,C=T)(T=I,C=C,R=1)(G=F,@=70)(T=I,D=69.1,70.2,71.2)
69 + (T=I)(T=I,C=C,R=1)(T=I,D=68.1)
70 EXISTS (T=S)(T=I)(C=2)(T=G,D=67.0,67.1,68.0,68.1,71.1,71.2,
   72.0)
```

```
        71 SELECT (T=S,C=T)(T=I,C=T)(T=R,D=72.1)
        72 MERGE (T=R)(T=R,C=C,R=524288)()(T=R,D=84.1)
        73 ID (T=S)(T=S,D=74.2)
        74 MERGE (T=S,C=T)(T=S)(G=F,@=77)(T=S,D=74.1,77.1,78.1)
        75 MERGE (T=I,C=T)(T=I,C=C,R=1)(G=F,@=77)(T=I,D=76.1,77.2,78.2)
        76 + (T=I)(T=I,C=C,R=1)(T=I,D=75.1)
        77 EXISTS (T=S)(T=I)(C=2)(T=G,D=74.0,74.1,75.0,75.1,78.1,78.2,
           79.0)
        78 SELECT (T=S,C=T)(T=I,C=T)(T=R,D=79.1)
        79 MERGE (T=R)(T=R,C=C,R=524288)()(T=R,D=91.1)
        80 CONS (T=S)(T=I,C=C,R=4)(T=I,D=81.3)
        81 APPEND (T=S,C=C,R='NIL')(T=I,R=1,C=C)(T=I)(T=S,D=82.1)
        82 APPEND (T=S)(T=I,R=2,C=C)(T=I,C=C,R=1)(T=S,D=83.2)
        83 APPLY* (R='Q',T=P,C=C)(T=S)(T=I,C=C,R=5)(T=S,D=)
        84 SEND (T=R)(T=R,D=)
        85 SEND (T=R)(T=R,D=)
        86 SEND (T=I)(T=I,D=)
        87 SEND (T=I,C=C,R=1)(T=I,D=)
        88 REC (T=R)(T=R,D=89.1,91.2,90.1)
        89 SACK1 (T=R)(T=I)(T=I)(T=I,D=)
        90 SACK2 (T=R)(T=I)(T=I)(T=I,D=)
        91 + (T=R)(T=R)(T=R,D=92.1,93.1)
        92 ID (T=R)(T=R,D=96.1)
        93 EOS (T=R)(C=1)(T=G,D=95.0,96.1,96.2,97.1)
        94 CONS (T=S)(T=R,C=C,R=0)(T=R,D=95.1)
        95 MERGE (T=R)(T=R)(G=T,@=93)(T=R,D=96.2,97.1)
        96 + (T=R,C=F)(T=R,C=F)(T=R,D=95.2)
        97 ID (T=R,C=T)(T=R,D=99.3)
        98 CONS (T=S)(T=F,R='OUTFILE',C=C) (T=F,D=99.1)
        99 WRITE (T=F)(R='F(10,5)',T=C,C=C)(T=R)(T=F,D=)
PROC Q
         0 SEND (T=R)(T=R,D=)
         1 REC (T=R)(T=R,D=5.1,19.1,18.1,6.1)
         2 REC (T=R)(T=R,D=7.1,15.1,14.1,8.1)
         3 REC (T=I)(T=I,D=9.1,21.1,11.2)
         4 REC (T=I)(T=I,D=10.1,17.1,12.1,11.1)
```

```
 5 SACK1 (T=R)(T=I)(T=I)(T=I,D=)
 6 SACK2 (T=R)(T=I)(T=I)(T=I,D=)
 7 SACK1 (T=R)(T=I)(T=I)(T=I,D=)
 8 SACK2 (T=R)(T=I)(T=I)(T=I,D=)
 9 ACK (T=I)(T=I)(T=I)(T=I,D=)
10 ACK (T=I)(T=I)(T=I)(T=I,D=)
11 > (T=I)(T=I)(C=0)(T=G,D=12.1,86.1.D,21.1,20.1.D,17.1,16.1.D)
12 ID (T=I,C=T)(T=I,D=13.1)
13 CONS (T=I)(T=R,R=524288.,C=C)(T=R,D=84.1)
14 ID (C=F,T=R)(T=R,D=25.1,76.1)
15 EOS (T=R)(C=1)(T=G,D=16.0,16.2)
16 MERGE (T=G)(T=G,C=F)(G=T,@=15)(T=G,D=14.1,16.2.D)
17 ID (C=F,T=I)(T=I,D=22.1,70.1,42.1)
18 ID (C=F,T=R)(T=R,D=45.1,75.1)
19 EOS (T=R)(C=1)(T=G,D=20.0,20.2)
20 MERGE (T=G)(T=G,C=F)(G=T,@=19)(T=G,D=18.1,20.2.D)
21 ID (C=F,T=I)(T=I,D=24.1,77.1,71.1,65.1,44.1)
22 - (T=I)(T=I,R=1,C=C)(T=I,D=23.1)
23 ID (T=I)(T=I,D=37.1)
24 CONS (T=I)(T=I,C=C,R=0)(T=I,D=35.1)
25 ID (T=R)(T=R,D=29.1,30.1)
26 - (T=I,C=F)(T=I,C=C,R=1)(T=I,D=35.2)
27 > (T=I)(T=I,C=C,R=0)(T=G,D=30.1)
28 ID (T=G)(C=1)(T=G,D=35.0)
29 EOS (T=R)(T=G,D=26.1,28.1.D)
30 ID (T=R,C=F)(T=R,D=31.1,32.1,40.1)
31 EOS (T=R)(T=G,D=36.1.D,37.2)
32 ID (T=R)(T=R,D=33.1)
33 ID (T=R)(T=R,D=34.1)
34 ID (T=R,C=T)(T=R,D=62.1)
35 MERGE (T=I)(T=I)(G=T,@=28)(T=I,D=26.1,27.1)
36 ID (T=G)(C=1)(T=G,D=37.0)
37 MERGE (T=I)(T=I,C=F)(G=T,@=36)(T=I,D=38.1,39.1)
38 - (T=I)(T=I,C=C,R=1)(T=I,D=37.2)
39 > (T=I)(T=I,C=C,R=0)(T=B,D=41.1)
40 EOS (T=R)(T=B,D=41.2)
41 | (T=B)(T=B)(T=G,D=34.1)
```

```
42 - (T=I)(T=I,R=1,C=C)(T=I,D=43.1)
43 ID (T=I)(T=I,D=57.1)
44 CONS (T=I)(T=I,C=C,R=0)(T=I,D=55.1)
45 ID (T=R)(T=R,D=49.1,50.1)
46 - (T=I,C=F)(T=I,C=C,R=1)(T=I,D=55.2)
47 > (T=I)(T=I,C=C,R=0)(T=G,D=50.1)
48 ID (T=G)(C=1)(T=G,D=55.0)
49 EOS (T=R)(T=G,D=46.1,48.1,D)
50 ID (T=R,C=F)(T=R,D=51.1,52.1,60.1)
51 EOS (T=R)(T=G,D=56.1,D,57.2)
52 ID (T=R)(T=R,D=53.1)
53 ID (T=R)(T=R,D=54.1)
54 ID (T=R,C=T)(T=R,D=62.2)
55 MERGE (T=I)(T=I)(G=T,@=48)(T=I,D=46.1,47.1)
56 ID (T=G)(C=1)(T=G,D=57.0)
57 MERGE (T=I)(T=I,C=F)(G=T,@=56)(T=I,D=58.1,59.1)
58 - (T=I)(T=I,C=C,R=1)(T=I,D=57.2)
59 > (T=I)(T=I,C=C,R=0)(T=B,D=61.1)
60 EOS (T=R)(T=B,D=61.2)
61 | (T=B)(T=B)(T=G,D=54.1)
62 * (T=R)(T=R)(T=R,D=63.1,64.1)
63 ID (T=R)(T=R,D=67.1)
64 EOS (T=R)(C=1)(T=G,D=66.D,67.1,67.2,68.1)
65 CONS (T=I)(T=R,C=C,R=0)(T=R,D=66.1)
66 MERGE (T=R)(T=R)(G=T,@=64)(T=R,D=67.2,68.1)
67 + (T=R,C=F)(T=R,C=F)(T=R,D=66.2)
68 ID (T=R,C=T)(T=R,D=69.1)
69 / (T=R)(T=I,R=2,C=C)(T=R,D=82.1)
70 + (T=I)(T=I,R=1,C=C)(T=I,D=78.1)
71 CONS (T=I)(T=I,C=C,R=4)(T=I,D=72.3)
72 APPEND (T=S,C=C,R='NIL')(T=I,R=1,C=C)(T=I)(T=S,D=73.1)
73 APPEND (T=S)(T=I,R=2,C=C)(T=I,C=C,R=I)(T=S,D=74.2)
74 APPLY* (R='Q',T=P,C=C)(T=S)(T=I,C=C,R=5)(T=S,D=)
75 SEND (T=R)(T=R,D=)
76 SEND (T=R)(T=R,D=)
77 SEND (T=I)(T=I,D=)
78 SEND (T=I)(T=I,D=)
```

```
       79 REC (T=R)(T=R,D=80.1,82.2,81.1)
       80 SACK1 (T=R)(T=I)(T=I)(T=I,D=)
       81 SACK2 (T=R)(T=I)(T=I)(T=I,D=)
       82 MERGE (T=R)(T=R)(G=T,@=83)(T=R,D=83.1,84.2)
       83 EOS (T=R)(C=1)(T=G,D=82.0)
       84 MERGE (T=R)(T=R)()(T=R,D=85.1,0.1)
       85 EOS (T=R)(C=1)(T=G,D=86.0,86.2)
       86 MERGE (T=G)(C=F,T=G)(G=T,@=85)(T=G,D=84.0,86.2.D)
PROC P
        0 SEND (T=R)(T=R,D=)
        1 REC (T=R)(T=R,D=5.1,20.1,19.1,6.1)
        2 REC (T=R)(T=R,D=7.1,17.1,16.1,8.1)
        3 REC (T=I)(T=I,D=9.1,14.1,11.2)
        4 REC (T=I)(T=I,D=10.1,15.1,12.1,11.1)
        5 SACK1 (T=R)(T=I)(T=I)(T=I,D=)
        6 SACK2 (T=R)(T=I)(T=I)(T=I,D=)
        7 SACK1 (T=R)(T=I)(T=I)(T=I,D=)
        8 SACK2 (T=R)(T=I)(T=I)(T=I,D=)
        9 ACK (T=I)(T=I)(T=I)(T=I,D=)
       10 ACK (T=I)(T=I)(T=I)(T=I,D=)
       11 > (T=I)(T=I)(C=0)(T=G,D=12.1,87.1.D,21.1.D,18.1.D,15.1,14.1)
       12 ID (T=I,C=T)(T=I,D=13.1)
       13 CONS (T=I)(T=R,R=524288.,C=C)(T=R,D=85.1)
       14 ID (C=F,T=I)(T=I,D=22.1,78.1)
       15 ID (C=F,T=I)(T=I,D=22.2,72.1,71.1,66.1,44.1,25.1)
       16 ID (C=F,T=R)(T=R,D=26.1,77.1)
       17 EOS (T=R)(C=1)(T=G,D=18.0,18.2)
       18 MERGE (T=G)(T=G,C=F)(G=T,@=17)(T=G,D=16.1,18.2.D)
       19 ID (C=F,T=R)(T=R,D=46.1,76.1)
       20 EOS (T=R)(C=1)(T=G,D=21.0,21.2)
       21 MERGE (T=G)(T=G,C=F)(G=T,@=20)(T=G,D=19.1,21.2.D)
       22 - (T=I)(T=I)(T=I,D=23.1)
       23 + (T=I)(T=I,R=1,C=C)(T=I,D=24.1,43.1)
       24 ID (T=I)(T=I,D=38.1)
       25 CONS (T=I)(T=I,C=C,R=0)(T=I,D=36.1)
       26 ID (T=R)(T=R,D=30.1,31.1)
```

```
27 - (T=I,C=F)(T=I,C=C,R=1)(T=I,D=36.2)
28 > (T=I)(T=I,C=C,R=0)(T=G,D=31.1)
29 ID (T=G)(C=1)(T=G,D=36.0)
30 EOS (T=R)(T=G,D=27.1,29.1,D)
31 ID (T=R,C=F)(T=R,D=32.1,33.1,41.1)
32 EOS (T=R)(T=G,D=37.1,D,38.2)
33 ID (T=R)(T=R,D=34.1)
34 ID (T=R)(T=R,D=35.1)
35 ID (T=R,C=T)(T=R,D=63.1)
36 MERGE (T=I)(T=I)(G=T,@=29)(T=I,D=27.1,28.1)
37 ID (T=G)(C=1)(T=G,D=38.0)
38 MERGE (T=I)(T=I,C=F)(G=T,@=37)(T=I,D=39.1,40.1)
39 - (T=I)(T=I,C=C,R=1)(T=I,D=38.2)
40 > (T=I)(T=I,C=C,R=0)(T=B,D=42.1)
41 EOS (T=R)(T=B,D=42.2)
42 | (T=B)(T=B)(T=G,D=35.1)
43 ID (T=I)(T=I,D=58.1)
44 - (T=I)(T=I,R=1,C=C)(T=I,D=45.1)
45 ID (T=I)(T=I,D=56.1)
46 ID (T=R)(T=R,D=50.1,51.1)
47 - (T=I,C=F)(T=I,C=C,R=1)(T=I,D=56.2)
48 > (T=I)(T=I,C=C,R=0)(T=G,D=51.1)
49 ID (T=G)(C=1)(T=G,D=56.0)
50 EOS (T=R)(T=G,D=47.1,49.1,D)
51 ID (T=R,C=F)(T=R,D=52.1,53.1,61.1)
52 EOS (T=R)(T=G,D=57.1,D,58.2)
53 ID (T=R)(T=R,D=54.1)
54 ID (T=R)(T=R,D=55.1)
55 ID (T=R,C=T)(T=R,D=63.2)
56 MERGE (T=I)(T=I)(G=T,@=49)(T=I,D=47.1,48.1)
```

```
57 ID (T=G)(C=1)(T=G,D=58.0)
58 MERGE (T=I)(T=I,C=F)(G=T,@=57)(T=I,D=59.1,60.1)
59 - (T=I)(T=I,C=C,R=1)(T=I,D=58.2)
60 > (T=I)(T=I,C=C,R=0)(T=B,D=62.1)
61 EOS (T=R)(T=B,D=62.2)
62 | (T=B)(T=B)(T=G,D=55.1)
63 * (T=R)(T=R)(T=R,D=64.1,65.1)
64 ID (T=R)(T=R,D=68.1)
65 EOS (T=R)(C=1)(T=G,D=67.0,68.1,68.2,69.1)
66 CONS (T=I)(T=R,C=C,R=0)(T=R,D=67.1)
67 MERGE (T=R)(T=R)(G=T,@=65)(T=R,D=68.2,69.1)
68 + (T=R,C=F)(T=R,C=F)(T=R,D=67.2)
69 ID (T=R,C=T)(T=R,D=70.1)
70 / (T=P)(T=I,R=2,C=C)(T=R,D=83.1)
71 + (T=I)(T=I,R=1,C=C)(T=I,D=79.1)
72 CONS (T=I)(T=I,C=C,R=4)(T=I,D=73.3)
73 APPEND (T=S,C=C,R='NIL')(T=I,R=1,C=C)(T=I)(T=S,D=74.1)
74 APPEND (T=S)(T=I,R=2,C=C)(T=I,C=C,R=1)(T=S,D=75.2)
75 APPLY* (R='P',T=P,C=C)(T=S)(T=I,C=C,R=5)(T=S,D=)
76 SEND (T=R)(T=R,D=)
77 SEND (T=R)(T=R,D=)
78 SEND (T=I)(T=I,D=)
79 SEND (T=I)(T=I,D=)
80 REC (T=R)(T=R,D=81.1,83.2,82.1)
81 SACK1 (T=R)(T=I)(T=I)(T=I,D=)
82 SACK2 (T=R)(T=I)(T=I)(T=I,D=)
83 MERGE (T=R)(T=R)(G=T,@=84)(T=R,D=84.1,85.2)
84 EOS (T=R)(C=1)(T=G,D=83.0)
85 MERGE (T=R)(T=R)()(T=R,D=86.1,0.1)
86 EOS (T=R)(C=1)(T=G,D=87.0,87.2)
87 MERGE (T=G)(C=F,T=G)(G=T,@=86)(T=G,D=85.0,87.2,D)
```

```
//E327MULT  JOB  I4987,CPLTJCL,MSGLEVEL=(1,1)
/*JOBPARM  LINES=15
//DFC  EXEC  PGM=DFC,REGION=356K,TIME=(,10),PARM=('ISA(-14K)/',
//    'CODE')
//STEPLIB  DD DSN=A.I4987.DFC.NEWCOM2,DISP=SHR
//SYSPRINT  DD  SYSOUT=A
//CODEFIL DD DSN=J.I4987.DFC.CODE1,UNIT=DISK,
//  DISP=(NEW,CATLG,DELETE),
//  DCB=(LRECL=80,BLKSIZE=800,RECFM=FB,BUFNO=1),
//  SPACE=(TRK,(10,1),RLSE)
//LASTFIL DD DSN=J.I4987.CODE.MULT,UNIT=DISK,
//    DCB=(LRECL=80,BLKSIZE=800,RECFM=FB,BUFNO=1),
//    DISP=(NEW,CATLG,DELETE),SPACE=(TRK,(10,2),RLSE,CONTIG)
//SYSTEST  DD  SYSOUT=A
//SYSDUMP  DD SYSOUT=A
//PLIDUMP  DD  SYSOUT=A,DCB=BLKSIZE=133
//SYSIN  DD *
 PROCEDURE MULT BEGIN
   REAL ARRAY AXX(1:10), AX2(1:10), AX3(1:10);
   REAL Z;
   INTEGER NPL;
   INTEGER STREAM S;
   REAL STREAM XX, X2, X3, R;
   FILE INFILE, OUTFILE;
   PROCEDURE P(IN(K, NPL, XX, X2), OUT(X3)) BEGIN
     INTEGER K, NPL, M;
     REAL X;
     REAL STREAM XX, X2, X3, Y;
     IF K > NPL
       THEN X3 := 524288.
```

```
    ELSE BEGIN
      M := NPL - K + 1;
      X := SUM(IN(SUBSTM(IN(XX,0,M)) *
                  SUBSTM(IN(X2,K-1,M)))) / 2;
      P(IN(K+1, NPL, XX, X2), OUT(Y));
      X3 := CONS(IN(X, Y))
      END
  END;
PROCEDURE Q(IN(K, NPL, XX, X2), OUT(R)) BEGIN
  INTEGER K, NPL;
  REAL STREAM XX, X2, R, Y;
  REAL X;
  IF K > NPL
    THEN R := 524288.
    ELSE BEGIN
      X := SUM(IN(SUBSTM(IN(XX,0,K-1)) *
                  SUBSTM(IN(X2,0,K-1)))) / 2;
      Q(IN(K+1, NPL, XX, X2), OUT(Y));
      R := CONS(IN(X, Y))
      END
  END;
INPUT NPL  FILE=INFILE FORMAT=I(2);
STREAM INPUT XX  FILE=INFILE FORMAT=F(7,1);
S := SCREATE(IN(1,NPL,1));
X2 := S;
P(IN(1, NPL, XX, X2), OUT(X3));
BUF XX, AXX;
BUF X2, AX2;
BUF X3, AX3;
XX := UNBUF(IN(AXX));
X2 := UNBUF(IN(AX2));
X3 := UNBUF(IN(AX3));
Q(IN(1, NPL, XX, X2), OUT(R));
Z := SUM(IN(X3 + R));
OUTPUT Z  FILE=OUTFILE FORMAT=F(10,5)
END
```

```
//STEP2 EXEC PGM=IEBPTPCH,COND=(0,EQ,DFC)
//SYSPRINT DD SYSOUT=A
//SYSUT1 DD DSN=J.I4987.DFC.CODE1,DISP=(OLD,DELETE)
//SYSUT2 DD SYSOUT=A
//SYSIN DD *
 PRINT MAXFLDS=1
 TITLE ITEM=('MULT')
 RECORD FIELD=(80)
/*
//DEL DD DSN=J.I4987.CODE.MULT,DISP=(OLD,DELETE)
//XS1 EXEC PGM=IEFBR14,COND=(0,NE,DFC)
//DD1 DD DSN=J.I4987.DFC.CODE1,DISP=(OLD,DELETE)
//ASM EXEC PGM=GO,REGION=100K,TIME=(,10),COND=(0,NE,DFC)
//* DD STATEMENTS FOR ASSEMBLER STEP
//STEPLIB DD DSN=P.I4119.ASMLOAD,DISP=SHR
//SYSIN DD DSN=J.I4987.CODE.MULT,DISP=(OLD,DELETE)
//SYSPRINT DD SYSOUT=A
//PLIDUMP DD SYSOUT=A,DCB=BLKSIZE=133
//SYSUDUMP DD SYSOUT=A
//MARC DD DSN=J.I4987.ASM.MULT,UNIT=DISK,DISP=(NEW,CATLG,DELETE),
//  SPACE=(TRK,(10,1),RLSE),
// DCB=(DSORG=DA,LRECL=252,BLKSIZE=252,RECFM=F)
//S1  EXEC  PGM=GO,REGION=550K,COND=(0,NE,DFC),PARM='NOSTAE,NOSPIE'
//* DD STATEMENTS FOR THE SIMULATOR STEP
//STEPLIB DD DSN=P.I4987.DFS.NEWSIM,DISP=SHR
//SYSPRINT  DD  SYSOUT=A
//ETC  DD  SYSOUT=A,DCB=(RECFM=FB,LRECL=132,BLKSIZE=132)
//PINFO DD DUMMY,DCB=(RECFM=F,LRECL=132,BLKSIZE=132)
//PFRNG DD DUMMY,DCB=(RECFM=F,LRECL=132,BLKSIZE=132)
//PINST DD DUMMY,DCB=(RECFM=F,LRECL=132,BLKSIZE=132)
//PLIDUMP  DD  SYSOUT=A
//SNAPDUMP  DD  SYSOUT=A
//SYSUDUMP DD SYSOUT=A
```

```
//SYSIN  DD  *
TRACE='1'B, FINAL='0'B;
13,1500,1,0,100,1
40000,60000,13000,8000,16000
3,3,1,2,40
6,1,5,5,1
1,1,1,1,1
8,8,10,8,5
9,3,3,3,3
3,9,9,3,9
3,1,1,4,4
4,11,1,4,15
20,5,15,15,5
5,5
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
1,1
02000001000000205242880

//MARC DD DSN=J.I4987.ASM.MULT,DISP=(OLD,KEEP)
```

APPENDIX C. NOTATION

The notation used in this dissertation is described in this appendix for a hypothetical node N.

Three main aspects of a node N might need to be denoted:

1) the type of SMD Petri net constructed from N used to compute an attribute of N: Ns or Np type Petri nets,
2) for a conditional node N, the choice of using only the true alternative (${}^{t}N$) or the false alternative (${}^{f}N$), and/or
3) for a node N containing portions of two streamed computations, the isolation of the streamed computation providing input values (N^{in}) or computing output values (N^{out}).

For the first aspect, Ns-type Petri nets have an additional designation as to whether a previously reduced node N' internal to N is to have a nodal firing time of $t_1(N')$ (denoted as Ns_1) or $t_2(N')$ (denoted as Ns_2).

Furthermore, these designations may be combined to properly specify the type of node being analyzed; e.g., ${}^{f}Np^{out}$ specifies that an Np-type Petri net was constructed from the streamed conditional node N using the false alternative only and only the portion involved with the construction of the output values.